SOFTWARE AND ALGORITHMS FOR LARGE-SCALE SEISMIC INVERSE PROBLEMS

A Dissertation
Presented to
The Academic Faculty

By

Philipp A. Witte

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

May 2020

SOFTWARE AND ALGORITHMS FOR LARGE-SCALE SEISMIC INVERSE PROBLEMS

Approved by:

Dr. Felix J. Herrmann, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Zhigang Peng
School of Earth and Atmospheric Sciences
Georgia Institute of Technology

Dr. Justin Romberg
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: February 19, 2020

To my wonderful parents

ACKNOWLEDGEMENTS

Above all, I would like to thank my adviser Professor Felix J. Herrmann for six fantastic years at SLIM and the opportunity to work among such an exceptional group of people. The inspiring work environment at SLIM and the close professional and personal relationships between me, my colleagues and you is what made my time as a Ph.D. student so productive and enjoyable. I am very grateful for everything I learned from you, the many opportunities for traveling around the world and for the freedom to pursue my research interests.

I would like to thank Professor Edmond Chow, Professor Richard Vuduc, Professor Zhigang Peng and Professor Justin Romberg for serving on my Ph.D. committee at Georgia Tech and for reviewing my thesis. Thank you for your valuable input and for taking time out of your busy schedules. Also, many thanks to my former Ph.D. committee at the University of British Columbia, namely Professor Eldad Haber, Professor Michael Friedlander and Assistant Professor Mark Schmidt, who accompanied me during the first years of my Ph.D. and throughout my candidacy.

I would like to acknowledge all of my current and former colleagues at SLIM, many of whom have become close personal friends over the years. Being able to collaborate closely with some of the smartest people I have ever met is what made my time at SLIM so valuable and enriching. I especially want to thank Mathias Louboutin, with whom I collaborated on many of my projects and whose deep knowledge in mathematics would always complement my geophysical background and increase the productivity of the both of us. I would also like to thank my colleagues from the Imperial College of London, Fabio Luporini, Navjot Kukreja and Michael Lange, all of whom I have worked with on multiple publications and whose work on Devito forms the basis of much of the research presented in this thesis.

Many thanks to Professor Gerard J. Gorman, with whom I closely worked together on Devito-related topics and who co-authored multiple of my papers. If it had not been for

the close collaboration between your group and SLIM, much of the work in this thesis would not have been possible. Thank you for hosting me and Mathias in London and for all the other fun get-togethers and hackathons in Denver, Anaheim and elsewhere around the world.

I would like to thank Dr. Henryk Modzelewski for his indispensable technical support over all my years at SLIM. I cannot think of anyone else with such a deep knowledge in all things computer-related and I am grateful for all your software contributions and the amount of patience you had with us over the years. Also many thanks to Charles Jones, who often provided much needed new perspectives and who saved me more than once during AWS-related problems with his deep knowledge in computing. It was your support that largely contributed to the success of my work on cloud computing. Many thanks also to your colleagues at Osokey James Selvage and Joseph Nicholson, who supported and promoted this work as well.

I owe much gratitude to Dr. Sverre Brandsberg-Dahl and his colleagues at Microsoft; Alexander Morris, Kadri Umay, Hussein Shel, Steven Roach and Evan Burness. Their hard work and support made it possible for me to apply my research to challenging real world problems and learn much about cloud computing in the process. Likewise, I would like to thank Dr. Kenton Prindle and his colleagues at Google for collaborating with me and my colleagues at the SEG cloud workshop and for giving us the opportunity to showcase our research on their platform.

As much of the work in this thesis builds upon existing software and utilizes a range of plotting and visualization packages, I would like to acknowledge the developers of Python and the Julia language, as well as the authors of Matplotlib/PyPlot, SeismicUnix, OpendTect, Inkscape, SegyIO.jl, JOLI.jl, PyCall.jl, Dierckx.jl, Flux.jl, equinor/segyio, boto3, Batch Shipyard and Docker.

Finally, I would like to acknowledge the institutions and grants that have provided funding over the course of my studies. These include the Natural Sciences and Engineering Re-

search Council of Canada Collaborative Research and Development Grant (DNOISE II), members of the SINBAD consortium, as well as the Georgia Research Alliance and the Georgia Institute of Technology.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

General usage of acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CFL	Courant–Friedrichs–Lewy
CG	Conjugate Gradient
CNN	Convolutional Neural Network
DFT	Discrete Fourier Transform
DSL	Domain Specific Language
FD	Finite Differences
EC2	Elastic Compute Cloud
FWI	Full Waveform Inversion
GCP	Google Cloud Platform
GD	Gradient Descent
HT	Hyper Threading
ISIC	Inverse Scattering Imaging Condition
JSON	Java Script Object Notation
JUDI	Julia Devito Inversion Framework
LS-RTM	Least Squares Reverse-Time Migration
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
RTM	Reverse-Time Migration
RVL	Rice Vector Library
ULFM	User Level Failure Mitigation
S3	Simple Storage System
SDK	Software Development Kit
SPG	Spectral Projected Gradient
SPLS-RTM	Sparsity Promoting Least Squares Reverse-Time Migration
SQS	Simple Queue Storage
TD	Time Domain

General usage of mathematical symbols

$\delta\mathbf{d}$	Linearized seismic data from single scattering
$\delta\mathbf{m}$	Seismic image/model perturbation
$\delta\mathbf{u}$	Linearized acoustic wavefield from single scattering
$\delta\mathbf{z}$	Acoustic impedance
$\mathbf{A}(\mathbf{m})$	Discretized acoustic wave equation
$\mathbf{C}$	Discrete curvelet transform
$\mathbf{d}_{\text{obs}}$	Observed seismic data
f	Temporal frequency
$\mathbf{F}$	Discrete Fourier transform
$\mathcal{F}(\mathbf{m},\mathbf{q})$	Acoustic forward modeling operator
$\mathbf{g}$	Gradient
$\mathbf{H}(\mathbf{m})$	Discretized Helmholtz operator
$\mathbf{J}$	Jacobian, partial derivative of modeling operator
$\mathbf{m}$	Squared slowness
$\mathbf{m}_0$	Squared slowness after applying a smoothing operator
$\mathbf{M}_l$, $\mathbf{M}_r$	Left- and right-hand preconditioners
$\mathbf{P}_r$	Receiver sampling operator
$\mathbf{P}_s$	Source sampling operator
$\mathcal{P}_\sigma$	Projection operator on to ℓ_2 ball with radius σ
$\mathbf{q}$	Seismic source
$\mathbf{R}$	Frequency restriction operator
$\mathbf{r}$	Residual between observed and predicted seismic data
S_λ	Soft thresholding function
t	Time
Δt	Time stepping interval
$\top$	Transpose
$\mathbf{u}$	Acoustic wavefield in the time domain
$\bar{\mathbf{u}}$	Acoustic wavefield in the frequency domain
$\mathbf{v}$	Adjoint wavefield in the time domain
$\bar{\mathbf{v}}$	Adjoint wavefield in the frequency domain
ω	Angular frequency

SUMMARY

Seismic imaging and parameter estimation are an import class of inverse problems with practical relevance in resource exploration, carbon control and monitoring systems for geohazards. The goal of seismic inverse problems is to image subsurface geological structures and estimate physical rock properties such as wave speed or density. Mathematically, this can be achieved by solving an optimization problem in which we minimize the mismatch between numerically modeled data and observed data from a seismic survey. As wave propagation through a medium is described by wave equations, seismic inverse problems involve solving a large number of partial differential equations (PDEs) during numerical optimization using finite difference modeling, making them computationally expensive. Additionally, seismic inverse problems are typically ill-posed, non-convex or ill-conditioned, thus making them challenging from a mathematical standpoint as well. Similar to the field of deep learning, this calls for software that is not only optimized for performance, but also enables geophysical domain specialists to experiment with algorithms in high-level programming languages and using different computing environments, such as high-performance computing (HPC) clusters or the cloud. Furthermore, they call for the adaption of dimensionality reduction techniques and stochastic algorithms to address computational cost from the algorithmic side. This thesis makes three distinct contributions to address computational challenges encountered in seismic inverse problems and to facilitate algorithmic development in this field. Part one introduces a large-scale framework for seismic modeling and inversion based on the paradigm of separation of concerns, which combines a user interface based on domain specific abstractions with a Python package for automatic code generation to solve the underlying PDEs. The modular code structure makes it possible to manage the complexity of a seismic inversion code, while matrix-free linear operators and data containers enable the implementation of algorithms in a fashion that closely resembles the underlying mathematical notation. The second contribution of

this thesis is an algorithm for seismic imaging, that addresses its high computational cost and large memory imprint through a combination of on-the-fly Fourier transforms, stochastic sampling techniques and sparsity-promoting optimization. The algorithm combines the best of both time- and frequency-domain inversion, as the memory imprint is independent of the number of modeled time steps, while time-to-frequency conversions avoid the need to solve Helmholtz equations, which involve inverting ill-conditioned matrices. Part three of this thesis introduces a novel approach for adapting the cloud for high-performance computing applications like seismic imaging, which does not rely on a fixed cluster of permanently running virtual machines. Instead, computational resources are automatically started and terminated by the cloud environment during runtime and the workflow takes advantage of cloud-native technologies such as event-driven computations and containerized batch processing. The performance and cost analysis shows that this approach is able to address current shortcomings of the cloud such as inferior resilience, while at the same time reducing operating cost up to an order of magnitude. As such, the workflow provides a strategy for cost effectively running large-scale seismic imaging problems in the cloud and is a viable alternative to conventional HPC clusters.

CHAPTER 1
INTRODUCTION

This thesis addresses current computational challenges arising in the context of seismic imaging and parameter estimation. These problems are concerned with imaging geological structures and estimating physical rock properties such as seismic velocities, density, impedance or porosity from seismic measurements that are recorded at the earth's surface [1, 2]. Seismic imaging and parameter estimation play an important role in today's oil and gas exploration for increasing success rates of drilling into reservoirs and thus decreasing the environmental impact of resource exploration. Furthermore they are used in monitoring systems for geohazards, as well as in carbon control and CO2 sequestration [3, 4]. As such, seismic exploration has the potential to play an important twofold role in addressing today's global challenges of an ever increasing demand for energy, as well as the need to remove CO2 from the atmosphere to limit the global rise in temperatures and help mitigate the effects of climate change.

Seismic images and parameters can be estimated from surface seismic measurements by mathematically formulating an inverse problem and by using numerical optimization techniques to minimize the mismatch between numerically predicted data and observed data from a seismic survey [5]. Objective functions and optimization algorithms for seismic imaging thus involve solutions of partial differential equations (PDEs), namely of wave equations in the time or frequency domain. In the general case, the relationship between the seismic measurements and the unknown parameters is nonlinear and objective functions are non-convex. Additionally, this class of inverse problems is typically ill-posed, as the information from the seismic data measured at the surface is not enough to uniquely determine the desired physical properties in the subsurface. Development of optimization algorithms for seismic inverse problems using (physics-inspired) constraints, regularization

or alternative misfit functions thus forms a large ongoing area of research [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

The second major challenge of seismic inverse problems is the high computational cost, as optimization algorithms involve repeatedly solving large-scale wave equations in two and three-dimensional domains using finite-difference modeling [17, 18, 19]. Equivalent to the training of neural networks, numerical optimization for seismic inverse problems is based on backpropagation and in principle requires storing the state variables of the forward problem in memory [5, 20, 21]. However, realistically sized seismic inverse problems are very high dimensional and waves have to be propagated for many wavelengths over thousands of time steps, making it impossible to store the state variables in memory. Methods such as domain decomposition or optimal checkpointing [22, 21] have to be used in practice to address the large size of the state variables (i.e. wavefields as a function of space and time), which lie in the order of up to 10^{12} floating point values. Furthermore, computations have to be carried out for a large number of individual seismic source experiments that form a seismic survey. This makes seismic inverse problems computationally expensive in a twofold way, as not only a large number of separate PDEs have to be solved during each iteration of an optimization algorithm, but every PDE solve itself is expensive in terms of memory and the required number of floating point operations (FLOPs) [23, 24].

The computational challenges of seismic inverse problems, combined with their mathematical properties of non-convexity and non-uniqueness, therefore require software that scales to peta- and eventually exascale problem sizes, while at the same time facilitating the implementation and development of complex inversion algorithms. The latter point is a prerequisite for enabling algorithmic innovation by geophysical domain experts, without requiring extensive knowledge in high-performance computing and optimizing compilers [25]. Second of all, the field of seismic inversion calls for the development and adaption of techniques from dimensionality reduction, compressive sensing and stochastic optimization to address computational challenges from an algorithmic point of view. As seismic

data in itself is highly redundant and subsurface images and parameters exhibit certain mathematical properties such as transform-domain sparsity [26, 27], many opportunities arise to exploit data and problem structures by means of recent algorithmic progress in convex optimization and signal processing [28, 29]. Finally, the high computational cost of seismic inverse problems makes access to high-performance computing (HPC) clusters inevitable for working on relevant problem sizes. However, these computational resources are only available to a very limited number of academic and research institutions, as well as to very few commercial companies. Many researchers are thus eliminated from making meaningful contributions to seismic inverse problems, solely due to the lack of access to compute. The recent rise of cloud computing offers an opportunity to democratize access to computational resources, but many open questions remain how these resources can be used efficiently for HPC applications with regard to cost, resilience, time-to-solution and software deployment [30, 31, 32, 33].

This thesis makes three distinct contributions regarding current challenges in seismic inverse problems, as discussed here:

- Chapter 2 (published in [34]) introduces a framework in the Julia programming language [35] for implementing seismic inversion algorithms using abstract high-level linear algebra expressions intended to increase the productivity of domain experts. The framework uses Devito, a Python package for automatic generation of optimized C code from symbolic Python expressions, to solve the underlying wave equations [19, 36, 37]. The framework thus represents a vertical integration of modern code generation and compiler tools into a domain-specific framework for seismic inverse problems that facilitates algorithmic innovation and reproducability, while at the same time delivering the necessary performance to work on relevant problem sizes.

- Chapter 3 (published in [38]) presents an algorithm for seismic imaging using on-the-fly discrete Fourier transforms (DFTs) and ideas from compressed sensing to address

the high memory- and computational cost of wave-equation based imaging. By combining time-to-frequency conversion methods with random sampling and sparsity-promoting minimization, I arrive at an algorithm whose memory requirements do not depend on the number of modeled time steps and whose computational cost is considerably reduced in comparison to full gradient methods. I compare the approach to alternative techniques such as optimal wavefield checkpointing and demonstrate that the proposed method arrives at acceptable solutions after a small limited number of data passes (epochs).

- Chapter 4 introduces a workflow for large-scale seismic imaging in the cloud using an event-driven and serverless approach, which does not rely on a densely connected cluster of permanently running compute instances. Instead, computational resources of the workflow are started and terminated automatically in response to events, thus eliminating idle instances and significantly reducing operating cost. The workflow is tested on several large-scale seismic data sets and I analyze conventional and cloud-specific performance metrics, such as scaling, turn-around time, cost and resilience.

The remainder of this introductory chapter is structured as follows; section 1.1 provides a brief introduction to seismic inverse problems and establishes the necessary terminology and mathematical notation. Section 1.2 develops the motives and objectives that will be addressed in this thesis by refining the challenges discussed in the previous paragraphs and by sketching out the proposed solutions. Section 1.3 provides the outline and structure of the thesis and section 1.4 lists the scientific contributions of the individual chapters.

1.1 Seismic inverse problems

1.1.1 The forward problem

Contrary to global seismology, exploration seismology is based on the manual excitation of seismic sources, which trigger sound and/or elastic waves that travel through the subsur-

face [1, 2]. At geological interfaces, waves are scattered and reflected, causing parts of the wavefield to travel back to the surface, where it is recorded by an array of receivers (Figure 1.1). After a source has been fired, each receiver records relative pressure changes in the water as a function of time. Seismic data is traditionally plotted by arranging the seismogramms of each receiver next to each other, with the receiver number on the horizontal axis and time on the vertical axis (Figure 1.2a). More commonly though, seismic data is plotted as an image of a two-dimensional array, in which each column corresponds to a receiver (as a function of time) and the color intensity denotes the pressure changes (Figure 1.2b). In a seismic survey, the source is fired repeatedly as it moves across the survey area and the observed data that is collected at each source location is called a *shot record*. Mathematically, each shot record is denoted by vectors $\mathbf{d}_i^{\text{obs}} \in \mathbb{R}^{n_d}$, where $i = 1, ..., n_s$ is the source index ranging from 1 to the total number of source locations n_s (or alternatively as a single concatenated vector $\mathbf{d}_{\text{obs}}$). The dimension n_d of the data is given by the number of receivers n_r times the number of time samples n_t. Note that seismic data is mathematically represented by vectors, but the actual data volume is a five-dimensional array with the dimensions time, receiver x- and y-coordinate and source x- and y-coordinate.

The objective of seismic inversion is to recover a physical parametrization of the subsurface from the recorded seismic data. In the setting of inverse problems, this is achieved by minimizing the misfit between recorded data and data that is predicted using numerical modeling. The *forward problem* is defined as the computation of a predicted seismic shot record $\mathbf{d}_i^{\text{pred}} \in \mathbb{R}^{n_d}$ through a forward modeling operator $\mathcal{F}$:

$$\mathbf{d}_i^{\text{pred}} = \mathcal{F}(\mathbf{m}, \mathbf{q}_i). \tag{1.1}$$

The vector $\mathbf{m} \in \mathbb{R}^n$ denotes the (vectorized) parameterization of the subsurface model, namely the acoustic wave-speed or p-wave velocity. The scalar n is the total number of grid points in two or three spatial dimensions. The vector $\mathbf{q}_i \in \mathbb{R}^{n_t}$ represents the time

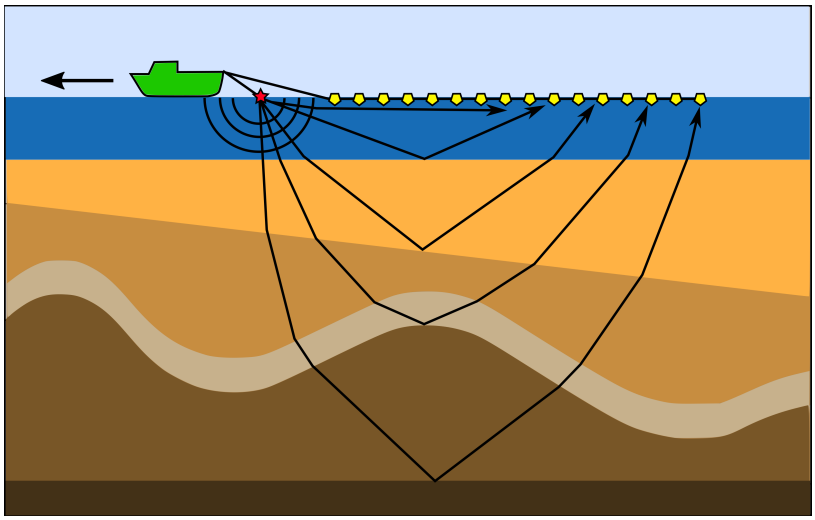


Figure 1.1: A seismic vessel excites acoustic waves that travel through the subsurface by firing a seismic source (red star). Reflected and scattered waves that travel back to the surface are recorded by an array of receivers (yellow pentagons) as a function of time and the experiment is repeated for a large number of different source locations.

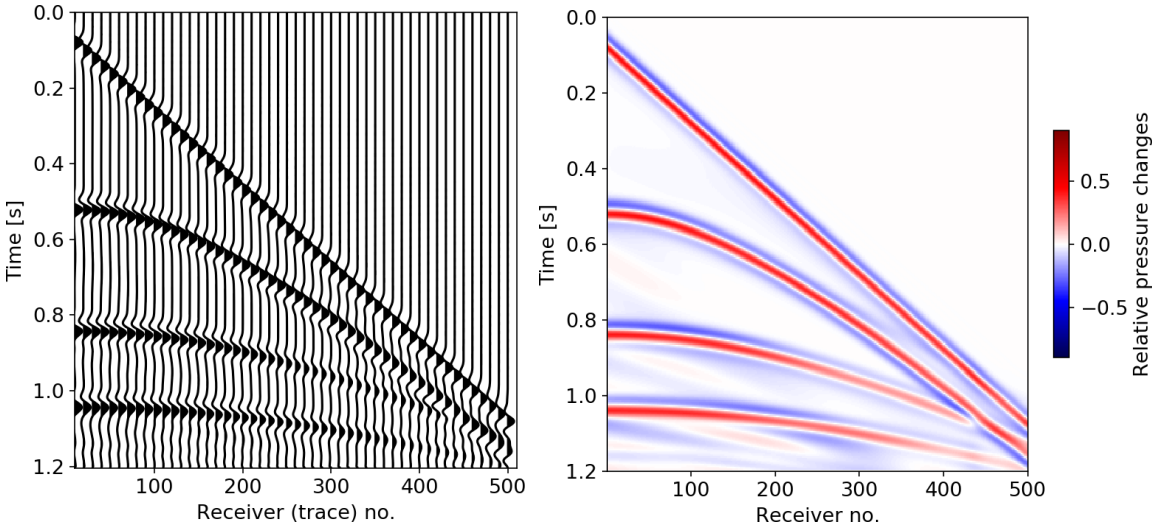


Figure 1.2: In a wiggle plot, a seismic shot record is plotted by horizontally arranging the seismogramms of individual receivers in ascending order (a). More commonly, a shot record is plotted as an image of a two-dimensional array, in which columns correspond to individual receivers (b).

signature of the seismic source, which is assumed to be a delta function in space with a constant radiation pattern, making it only a function of time, with n_t being the number of time samples. The forward modeling operator $\mathcal{F}$ is defined as the solution of a discretized wave equation for a given model and source:

$$\mathcal{F}(\mathbf{m}, \mathbf{q}_i) = \mathbf{P}_r \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^\top \mathbf{q}_i. \tag{1.2}$$

The matrix $\mathbf{P}_s \in \mathbb{R}^{n_s \times N}$ is the source projection operator, whose transpose (denoted by $\top$) injects the time dependent source signature at the current source location. Effectively, the source projection operator maps the (short) time-dependent vector $\mathbf{q}_i \in \mathbb{R}^{n_t}$ to a vector with the dimensions of the full time and spatially dependent wavefield $\mathbf{u} \in \mathbb{R}^N$, where $N = n \cdot n_t$ is the number of grid points times the number of time samples. Accordingly, $\mathbf{P}_r \in \mathbb{R}^{n_d \times N}$ is the receiver projection operator, which samples the time-dependent wavefield at the location of the receivers. Generally, this involves linear interpolation, as the receiver (and source) locations do not necessarily coincide with the computational grid. The operator $\mathbf{A}(\mathbf{m}) \in \mathbb{R}^{N \times N}$ represents the discretized, time-dependent wave equation:

$$\underbrace{\left(\mathbf{m} \frac{\partial^2}{\partial t^2} - \nabla^2 \right)}_{\mathbf{A}(\mathbf{m})} \mathbf{u} = \mathbf{P}_s^\top \mathbf{q}_i. \tag{1.3}$$

The operators $\frac{\partial^2}{\partial t^2} \in \mathbb{R}^{N \times N}$ and $\nabla^2 \in \mathbb{R}^{N \times N}$ denote the second temporal derivative and the Laplacian, both discretized using finite differences (FD) [see e.g. 39, in the context of seismic modeling]. Here, $\mathbf{A}(\mathbf{m})$ is written as a linear operator acting on the wavefield $\mathbf{u}$, which is correspondingly obtained by applying the inverse of the operator to the equation's right-hand side, i.e. $\mathbf{u}_i = \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^\top \mathbf{q}_i$. The seismic data is then obtained by applying the receiver restriction operator to the wavefield, namely $\mathbf{d}_i^{\text{pred}} = \mathbf{P}_r \mathbf{u}_i$. For solving the forward problem, the operator $\mathbf{A}(\mathbf{m})^{-1}$ is not inverted explicitly, but instead its matrix-vector product with the right-hand side is computed using finite-difference time-stepping

[detailed explanations presented in 40, 41].

1.1.2 The inverse problem

The goal of seismic inverse problems is the recovery of model parameters $\mathbf{m}$ from the set of observed seismic data $\mathbf{d}_i^{\text{obs}}$. The problem is most commonly formulated as a nonlinear (unconstrained) optimization problem in which the objective is to minimize the data misfit between the observed and predicted data [5, 9]:

$$\underset{\mathbf{m}}{\text{minimize}} \ \sum_{i=1}^{n_s} \phi\Big[\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i^{\text{obs}}\Big]. \tag{1.4}$$

Here, $\phi : \mathbb{R}^{n_d} \rightarrow \mathbb{R}$ is a (smooth) misfit function and typically chosen to be the least-squares misfit $\phi = \frac{1}{2}\|\cdot\|^2$. This version of the inverse problem is referred to as the reduced-state or adjoint-state formulation, but other formulations that are derived from the more generic constrained formulation exist as well [42, 10, 43]. In its reduced form (equation 1.4), the (nonlinear) seismic inverse problem is called *Full Waveform Inversion* (FWI), as the goal is the prediction of *all* waveforms in the seismic data (i.e. reflection, diffractions, refractions, turning waves) by inverting for the model parameters $\mathbf{m}$.

FWI is commonly solved using gradient-based optimization algorithms such as gradient descent, Gauss-Newton methods or Quasi-Newton methods [7, 9, 12, 16]. As mentioned, the gradient of equation 1.4 with respect to the model parameters $\mathbf{m}$ is calculated using the adjoint state method, also known as a reduced space method in the optimization literature [44]. For the case where ϕ is the least-squares misfit, the (adjoint-state) gradient is given by [45, 5]:

$$\mathbf{g} = \sum_{i=1}^{n_s} \left(\frac{\partial \mathcal{F}(\mathbf{m}, \mathbf{q}_i)}{\partial \mathbf{m}}\right)^{\top} \Big(\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i^{\text{obs}}\Big), \tag{1.5}$$

where $\mathbf{g} \in \mathbb{R}^n$ is the gradient of the FWI objective function and $\frac{\partial \mathcal{F}}{\partial \mathbf{m}} \in \mathbb{R}^{n_d \times n}$ is the partial derivative of the forward modeling operator with respect to $\mathbf{m}$, which is commonly known as the Jacobian and physically corresponds to linearized Born scattering [46]. Computing

the gradient of the FWI objective function thus involves solving a forward and adjoint (linearized) wave equation for every index i of the sum over the source locations, with n_s being in the order of 10^2 to 10^5. Furthermore, the relationship between $\mathbf{m}$ and the predicted data is nonlinear and the objective function is non-convex, so additional steps (regularization, constraints, etc.) need to be taken into account. A closely related variation of FWI with high practical importance is the linearized inverse problem, in which the Jacobian itself is used as the forward modeling operator.

1.1.3 Linearized inversion

In the linearized seismic inverse problem, the underlying assumption is that a piecewise constant/smooth model $\mathbf{m}$, such as the one plotted in Figure 1.3a, can be separated into a smooth component $\mathbf{m}_0 \in \mathbb{R}^n$ containing the low wavenumbers (Figure 1.3b) and a high wavenumber component $\delta\mathbf{m} \in \mathbb{R}^n$, containing the strong parameter contrasts (Figure 1.3c). Accordingly, the nonlinear forward modeling operator $\mathcal{F}(\mathbf{m}, \mathbf{q}_i)$ is approximated by the sum of $\mathcal{F}$ evaluated at $\mathbf{m}_0$ plus the action of the Jacobian on $\delta\mathbf{m}$:

$$\mathcal{F}(\mathbf{m}, \mathbf{q}_i) \approx \mathcal{F}(\mathbf{m}_0, \mathbf{q}_i) + \frac{\partial \mathcal{F}(\mathbf{m}_0, \mathbf{q}_i)}{\partial \mathbf{m}} \delta\mathbf{m} + \mathcal{O}(\delta\mathbf{m}^\top \delta\mathbf{m}). \tag{1.6}$$

The physical intuition behind this separation of scales, is that seismic data can be (in approximation) divided into contributions from transmitted waves that propagate through $\mathbf{m}_0$ (i.e. direct and turning waves), while reflections and scattered waves are caused by the model perturbations in $\delta\mathbf{m}$ [6]. The higher-order terms of equation 1.6 describe data that results from multiple scattering.

The Jacobian $\frac{\partial \mathcal{F}(\mathbf{m}, \mathbf{q}_i)}{\partial \mathbf{m}}$ (the same linear operator as in equation 1.5), is obtained by taking the derivative of equation 1.3 with respect to $\mathbf{m}$, which yields:

$$\frac{\partial \mathcal{F}(\mathbf{m}, \mathbf{q}_i)}{\partial \mathbf{m}} = -\mathbf{P}_r \mathbf{A}(\mathbf{m})^{-1} \mathrm{diag}\left(\frac{\partial \mathbf{A}(\mathbf{m})}{\partial \mathbf{m}} \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^\top \mathbf{q}_i \right). \tag{1.7}$$

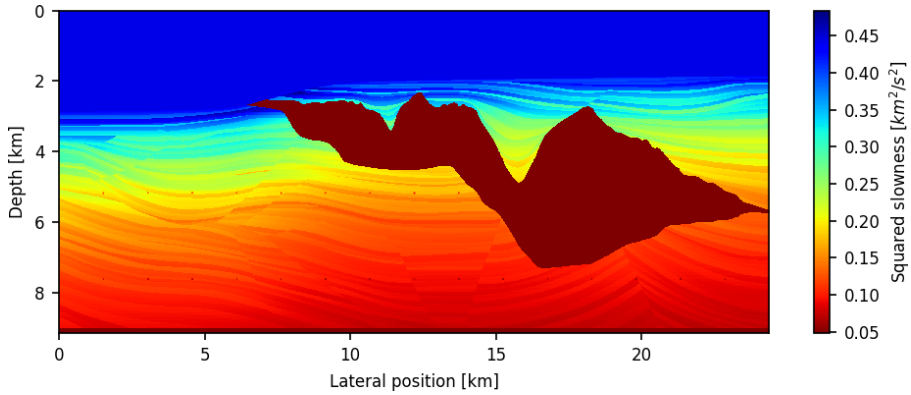
The Jacobian thus maps perturbations in the model to perturbations in the seismic data. Throughout the thesis, I denote the Jacobian by $\mathbf{J}$ to highlight the fact that in contrast to $\mathcal{F}$, it is a linear operator.

The model perturbation $\delta\mathbf{m}$ with respect to a smooth (background) model $\mathbf{m}_0$ is called the seismic image. Its practical value is that it provides structural information of the subsurface, i.e. the location of geological interfaces, which are associated with rapid changes in wave speed and/or density. Therefore, if a good approximation of $\mathbf{m}_0$ is available (e.g. from FWI), the seismic image can by obtained through linearized waveform inversion by solving the following linear least squares problem:

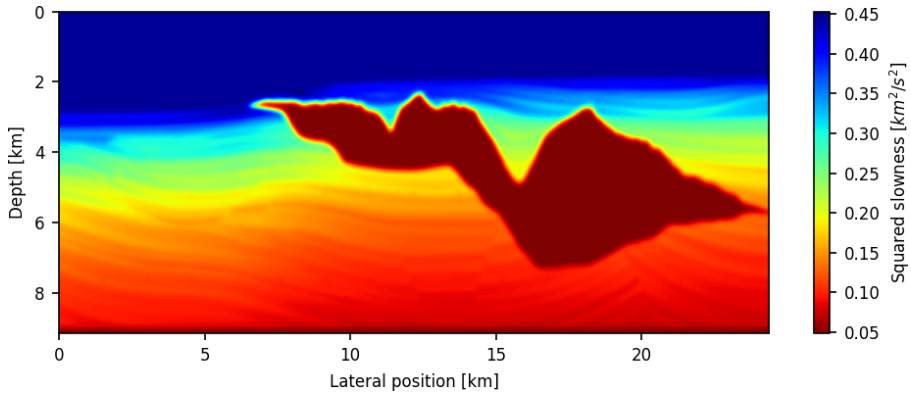
$$\underset{\delta\mathbf{m}}{\text{minimize}} \sum_{i=1}^{n_s} \frac{1}{2}\|\mathbf{J}\big(\mathbf{m}_0, \mathbf{q}_i\big)\ \delta\mathbf{m} - \delta\mathbf{d}_i^{\text{obs}}\|^2. \tag{1.8}$$

The vector $\delta\mathbf{d}_i^{\text{obs}} = \mathbf{d}_i^{\text{obs}} - \mathcal{F}(\mathbf{m}_0, \mathbf{q}_i)$ is the linearized observed data, i.e., seismic data in which ideally all effects of transmission waves and multiple scattering have been removed prior to the inversion. Linearized seismic inversion in the above form is referred to as *least squares reverse-time migration* (LS-RTM), as the data residual has to be back-propagated through time to compute the gradient of the objective function, which is the same technique used in deep learning [47, 48, 49, 50]. The term *migration* refers to the process mapping events (i.e. reflections) in the observed seismic data (Figure 1.4) to subsurface perturbations in the image space that caused them.

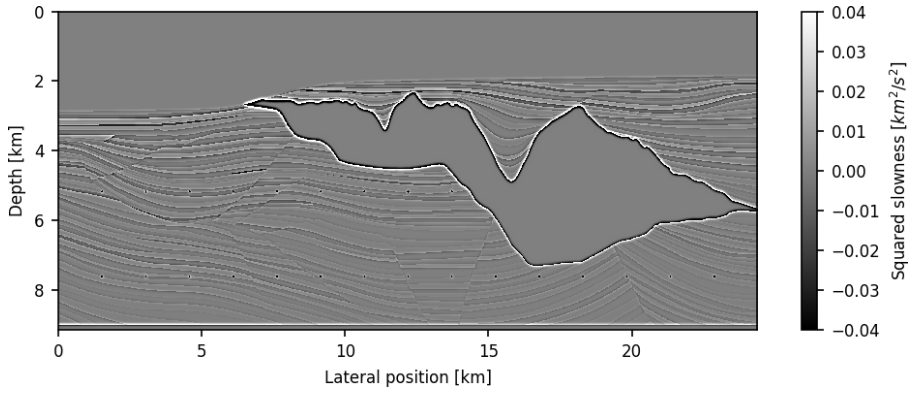
Having formalized the underlying mathematical problem and notation, the next section will point out current challenges encountered in seismic inversion and provide a brief overview of the literature. Furthermore, I formulate the research objectives of this thesis and their novelty in comparison to current approaches.



(a)



(b)



(c)

Figure 1.3: The Sigsbee 2A model is a synthetic p-wave velocity model (here in squared slowness). Blue denotes the water column and the dark red object represents a salt body. The background consists of sedimentary layers. Convolving the true model (a) with a Gaussian kernel yields a smooth background velocity (b), which is assumed to be known for seismic imaging. The objective of seismic imaging is the recovery of the high-contrast velocity perturbation (c) with respect to the background model (b). The perturbation (c) is called the seismic image and is conventionally plotted in grayscale. Oftentimes the colorbar is omitted, as images are normalized and rescaled for plotting purposes, causing magnitudes to lose their physical meaning.

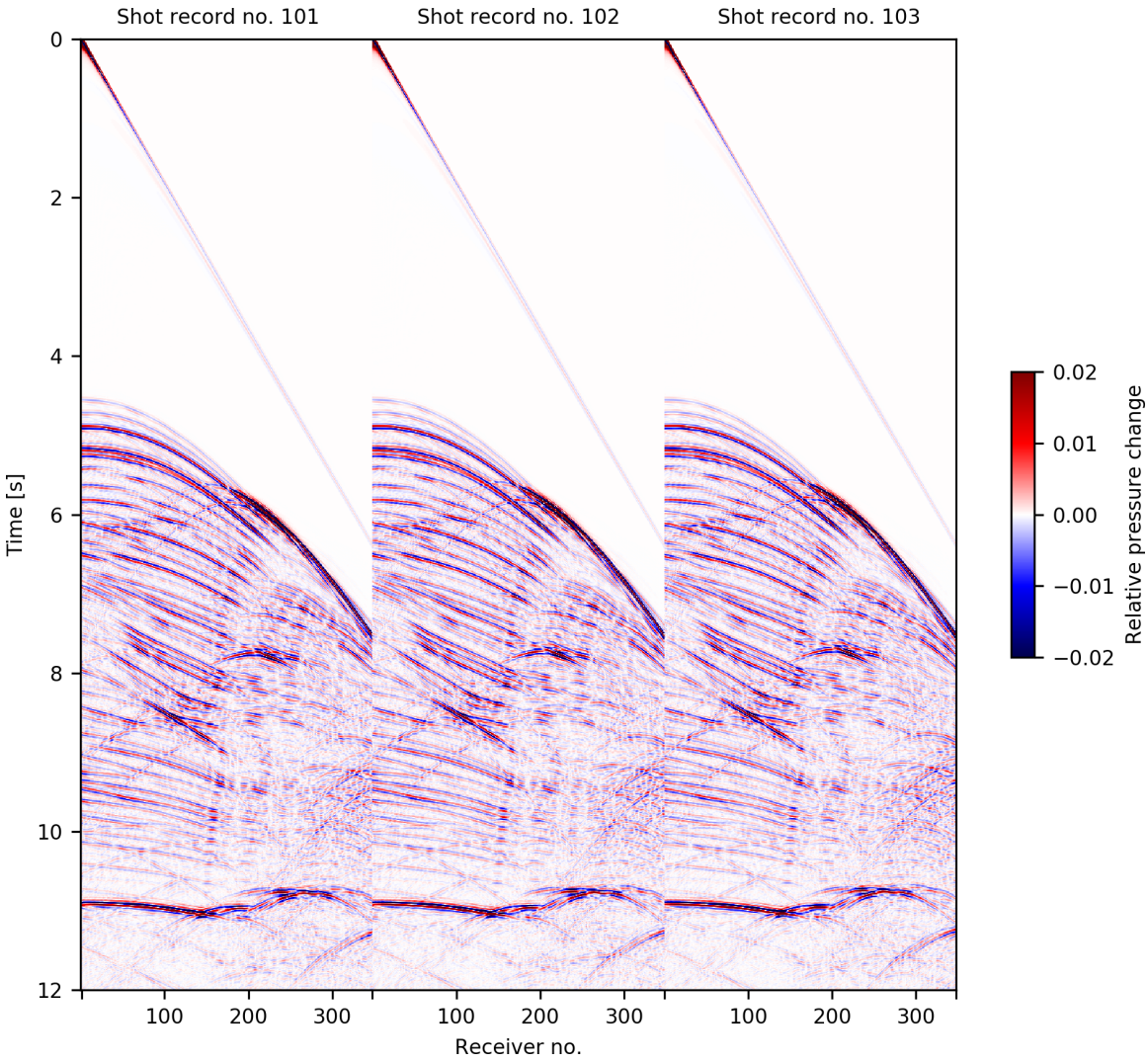


Figure 1.4: Observed seismic data that was computed for the Sigsbee 2A model (Figure 1.3a) using finite-difference modeling. The full data set consists of 500 shot records $\mathbf{d}_i^{\text{obs}}$ $(i = 1, ..., 500)$, of which three are plotted here side by side. Each shot record is a 2D array of dimensions 3001×348 (times samples $\times$ number of receivers). The objective of seismic imaging is to estimate the model perturbation $\delta\mathbf{m}$ (Figure 1.3c) from this seismic data.

1.2 Motives and objectives

1.2.1 Software for seismic inversion

As outlined in the introductory paragraph, challenges of nonlinear seismic inverse problems (i.e. FWI) arise from its non-convexity, as well as from physical limitations, such as the absence of low temporal frequencies in the data and incomplete illumination of the subsurface [8, 9]. In practice, it is therefore not easily possible to arrive at a satisfactory estimation of the unknown parameters $\mathbf{m}$, without going through multiple iterations of data processing, experimentation with different optimization algorithms and refinement of the initial starting value. Ideally, this processes is facilitated by software with high levels of user abstractions, which allows domain experts to experiment with problem formulations and algorithms. In the closely related field of deep learning, it is precisely the wide availability and popularity of domain-specific frameworks like PyTorch [51] and Tensorflow [52], which have stimulated algorithmic innovation and reproducibility by exposing functionalities through abstract user interfaces.

In the field of seismic inversion however, software for modeling and inversion is traditionally implemented in low-level languages such as Fortran or C, with a large amount of manual performance optimization [23, 53]. Codes are typically monolithic applications, making it challenging to modify existing software, as components of PDE solvers, parallelization and optimization are oftentimes not clearly separated. Furthermore, commercially developed software is almost exclusively proprietary and not available to the public. Open-source academic software packages on the other hand [e.g. 54, 55, 56] often trade performance for abstractions and flexibility, making it easier to modify code, but at the price of impaired performance. This dichotomy of proprietary low-level codes on the one side and non-optimized high-level research frameworks on the other side motivates the first objective of this thesis:

- The development of a framework for seismic modeling and inversion, which exposes functionalities required for seismic inversion in a high-level symbolic language, while at the same time providing the performance to work on realistic problem sizes. This objective will be achieved through a code design based on the separation of concerns and automatic performance optimizations. Furthermore, the objective is to make this software publicly available and to provide a series or reproducible examples and tutorials to facilitate algorithmic innovation, without sacrificing performance.

1.2.2 Scalable algorithms for seismic imaging

While the LS-RTM objective function for linearized inversion is quadratic and has in principle an analytic solution, the problem is generally inconsistent, as the observed data lies outside the range of the Jacobian (due to linearization errors) and contains noise. As the linear operators used to represent Jacobians and wave equations are too large to be formed explicitly, seismic imaging requires iterative matrix-free optimization algorithms like the nonlinear conjugate gradient (CG). However, full gradient methods like CG compute the gradient over the full sum in the objective function (equation 1.8), making these methods too expensive in practice, when only a small number of overall data passes (epochs) are affordable. This has motivated the development of dimensionality reduction techniques such as seismic source encoding [57, 58, 59, 60, 61, 62, 63] and the adaption of stochastic sampling and optimization algorithms [64, 65, 66, 67].

Another major challenge of seismic imaging is related to the large memory demand that is associated with computing the gradient using the adjoint state method. As backpropagation requires access to the forward state variables, forward modeled wavefields have to be stored in memory or on disk, but this approach is not feasible for problem sizes encountered in practice. Current approaches to circumvent this problem include writing wavefields to disk, wavefield checkpointing [22, 21] or the reconstruction of wavefields from the model

boundary [68]. These approaches decrease the required memory in exchange for additional computations, but overall the memory demand still grows with the number of time steps, i.e. with the recording length of seismic data. Alternatively, seismic imaging can be formulated in the frequency domain, where computations are separable over frequencies and backpropagation over all time steps is not required. However, solving Helmholtz equations for high-frequencies and large domains requires the usage of iterative Krylov solvers and the poor conditioning of the problem typically leads to slow convergence [e.g. 69, 70]. These challenges motivate the second objective of this thesis:

- The development of time-domain seismic imaging algorithms that overcome the curse of dimensionality by exploiting redundancy and sparsity of seismic data and images. Specifically, I am interested in decreasing the overall number of wave equations solves to a small number of passes through the data. Furthermore, the objective is the development of techniques whose memory requirements are (ideally) independent of the number of time steps and do not require solutions of Helmholtz equations. These objectives are achieved by combining existing approaches for sparsity-promoting seismic imaging in the frequency domain [26, 64], with time-to-frequency conversions based on on-the-fly Fourier transforms [71].

1.2.3 Adapting the cloud for seismic inversion

The high computational cost of seismic inverse problems makes high-performance computing clusters necessary to work on relevant problem sizes that are currently encountered in industrial applications. Major companies such as ExxonMobil, Petroleum Geo-Services (PGS) or BP operate their own HPC clusters, with maximum achievable performance in the order or petaflops [72, 73, 74]. While HPC systems are dominantly used in production, they also play an import role in research, as newly developed algorithms require validation and testing on realistic large-scale data sets. Gaining access to HPC resources is challenging for small and medium-sized companies, as well as for academic research groups due to

the large upfront and maintenance cost of these systems. Furthermore, research cycles typically consists of a development stage, followed by the validation/testing period and access to HPC resources are mostly required during the later stage only. In an academic setting, it is therefore challenging to run HPC clusters at full capacity or deploy large-scale cluster workloads, as long-running jobs are often treated with lower priority by job schedulers than smaller workloads. Additionally, purchasing HPC systems often involves compromises regarding the architecture and hardware choices, as users from different scientific backgrounds with varying demands typically access the same machine.

The recent rise of cloud computing offers the opportunity to address these challenges and to make HPC resources available to a broader community. Cloud providers such as Amazon Web Services (AWS), Azure or the Google Cloud Project (GCP) offer access to computational resources on a pay-as-you-go pricing model with no upfront cost. Users can choose from a large variety of hardware (i.e. various architectures, accelerators, etc.) and configure computational resources exactly as needed, while only paying for the time that resources were utilized. An open question that remains in this context, is how the cloud can be used most efficiently and cost effectively for HPC applications. Many performance evaluations and benchmarks of typical HPC workloads find that the cloud can generally not offer the same performance, resilience, bandwidth and latency as comparable on-premise HPC systems [31, 32, 75, 33]. This is especially problematic for many applications in computational science and engineering (CSE) and HPC, which are developed for densely connected clusters and are based on message passing (MPI) [76]. Codes of this type rely on close communications between computational resources and predictable behavior of the hardware, due to low fault tolerances. Using the cloud to replicate on-premise HPC systems to run software that was developed for classical clusters (*lift and shift*) is therefore not an ideal approach to adapt the cloud for HPC applications. Thus, the third objective of this thesis is:

- The development of a framework for seismic imaging in the cloud that is not based on the conventional lift and shift approach and does not rely on a densely connected cluster of virtual instances. This involves analyzing the structure of the underlying mathematical problem to identify possibilities in which it is possible to take advantage of novel cloud technologies such as object storage, containerized batch computing and event-driven computations. The objective is the development and application of such a *serverless* cloud framework for seismic imaging, as well as its analysis in terms of performance, resilience, turn-around time and cost.

1.3 Thesis outline

This thesis is divided into three main chapters 2–4, each of which is devoted to one of the objectives that were derived in the previous section. Each chapter follows the general structure of a technical journal article and begins with an introduction into the respective topic and a review of the current literature and state of the art.

Chapter 2 [34] presents a framework for seismic modeling and inversion in the Julia language based on the paradigms of separation of concerns and abstract user interfaces. The chapter contains a presentation of the software structure and justifies the design choices that were made for the user interfaces. These consists of matrix-free linear operators and abstract data containers, that enable the implementation of objective functions and algorithms following the mathematical notation presented in section 1.1. Matrix-vector products involving the solution of wave equations interface Devito to generate optimized stencil code for solving the underlying PDEs. The parallelization of the sum over the source locations is implemented in Julia and takes advantage of its built-in resilience, which allows that workloads of failed instances are re-processed by remaining workers. The majority of the chapter features a series of numerical case studies and demonstrates how the framework can be utilized to implement a variety of algorithms for linear and nonlinear seismic inverse problems in a symbolic fashion that closely resembles the mathematical notation.

Chapter 3 [38] introduces an algorithm for least squares seismic imaging using on-the-fly Fourier transforms and sparsity-promoting minimization. The theoretical section presents the derivation of forward-adjoint pairs of the linearized Born scattering operator with on-the-fly Fourier transforms and numerical modeling based on time-stepping. This enables the computation of monochromatic wavefields and images in the frequency domain using time-domain modeling codes. Subsequently, this chapter demonstrates that by adapting randomized sampling techniques inspired by compressive sensing and randomized linear algebra, it is possible to work with a small number of random temporal frequencies and seismic source locations in each iteration of the optimization algorithm. Overall, this leads to an algorithm whose memory is independent of the number of time steps, as gradients are computed for a small number of randomly selected frequencies, rather than as a sum of all frequencies or time steps. Furthermore, I demonstrate that the approach arrives at acceptable solutions with as few as two passes through the data and that results are qualitatively comparable to time-domain results without frequency subsampling. In the numerical examples section, I apply the algorithm to two large-scale 2D data sets and analyze the trade-off between varying the frequency versus the source batch size, as well as the role of the regularization parameter.

Chapter 4 proposes a novel approach for adapting the cloud for high-performance computing tasks. Specifically, I consider the application of seismic imaging on Amazon Web Services and introduce an alternative strategy to the common lift-and-shift approach, which leverages cloud technologies such as event-driven computations. These techniques are adapted to address cloud-related challenges such as the high operating cost that incurs if computational resources are idle and not used at full capacity, as well as inferior resilience compared to on-premise clusters. The chapter describes the different components of the workflow, which implement a serverless map-reduce model based on batch processing and event-driven computations. In contrast to the lift and shift approach, my workflow does not rely on a cluster of permanently running virtual machines, as compute instances are

launched and terminated automatically in response to *events*, such as the increase of an iteration count. In the performance analysis, I examine conventional performance metrics such as weak and strong scaling, as well as cloud-specific metrics such as cost, resilience and the influence of overhead from restarting instances to the time-to-solution. In the discussion, I illustrate advantages and shortcomings of the proposed approach and review software requirements that are necessary to run applications in the cloud in a serverless fashion.

1.4 Contributions

The following list summarizes the scientific contributions that are presented in this thesis:

- My first contribution is an open-source software framework in the Julia language for seismic modeling and inversion. The novelty is a combination of domain-specific abstractions that enable symbolic implementations of seismic inversion algorithm with techniques for automatic code generation and performance optimization to solve the underlying PDEs. My contribution consists of the design and the implementation of the Julia framework, including the API, parallelization, matrix-free linear operators and out-of-core data containers that allow working with large-scale data sets. My contribution further includes the Julia interface to Devito (whose API and compiler were developed by [41] and [37]), and the implementation of forward/adjoint wave equations in the context of the Julia framework. This work was published in the *Software and Algorithms* section of *Geophysics* [34], and as such includes code and instructions to reproduce the numerical examples. Furthermore, the software framework is featured in a tutorial series on full-waveform inversion in *The Leading Edge* [77].

- My second contribution is an algorithm for least squares imaging using on-the-fly Fourier transforms, which enables seismic imaging at a fraction of the computational cost of comparable approaches in the time domain. The novelty of this approach is the combination of time-to-frequency conversion methods with stochastic sampling and sparsity-promoting minimization, which yields an algorithm whose memory imprint is independent of the number of modeled time steps. My contribution includes the derivation of forward and adjoint operators for linearized Born scattering in the time and frequency domain for different physical parametrizations (velocity and impedance). Furthermore, I contribute insights into sparsity-promoting minimization techniques in the context of seismic imaging, including the trade-off between frequency versus source subsampling and the role of the thresholding parameter for problems in which only a limited number of data passes are computationally feasible. This work was published as a technical article in *Geophysics* [38] and includes open-source code to reproduce the numerical examples from the paper [78]. All examples of this chapter are implemented using my Julia inversion framework from chapter 2.

- My third contribution is the development and implementation of a serverless seismic imaging framework in the cloud. The novelty of this work is a strategy for deploying HPC applications to the cloud, which goes beyond the conventional lift and shift approach and does not rely on a cluster of permanently running virtual machines. My contribution includes the adaption of cloud services to implement generic iterative optimization algorithms as a collection of serverless tasks that are executed by the cloud environment. Each iteration of the optimization algorithm is essentially a map-reduce problem and I develop a serverless implementation of this model for the specific case where the (embarrassingly parallel) map part is expensive to compute and has a considerably longer time-to-solution than the reduce part. My contribution includes an implementation of this approach for seismic imaging on AWS and Azure, a performance analysis, as well as a case study on Azure using a large-scale

three-dimensional seismic data set. Remark: This work has been submitted to IEEE Transactions on Parallel and Distributed Systems and is currently under review. A preprint is available on arXiv [79] and an early version of this work was presented in [80].

REFERENCES

[1] R. E. Sheriff and L. P. Geldart, *Exploration seismology*. Cambridge University Press, 1995.

[2] O. Yilmaz, *Seismic data analysis: Processing, inversion, and interpretation of seismic data*. Society of Exploration Geophysicists, 2001.

[3] S. J. Baines and R. H. Worden, "Geological storage of carbon dioxide," *Geological Society, London, Special Publications*, vol. 233, no. 1, pp. 1–6, 2004.

[4] D. Lumley, "4D seismic monitoring of CO2 sequestration," *The Leading Edge*, vol. 29, no. 2, pp. 150–155, 2010.

[5] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, no. 8, p. 1259, 1984.

[6] J. Claerbout, *Earth Soundings Analysis: Processing Versus Inversion*. Blackwell Scientific Publications, 1992, ISBN: 9780865422100.

[7] R. G. Pratt, C. Shin, and G. J. Hick, "Gauss–Newton and full Newton methods in frequency–space seismic waveform inversion," *Geophysical Journal International*, vol. 133, no. 2, p. 341, 1998.

[8] W. Mulder and R.-E. Plessix, "Exploring some issues in acoustic full waveform inversion," *Geophysical Prospecting*, vol. 56, no. 6, pp. 827–841, 2008.

[9] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, WCC127–WCC152, Nov. 2009.

[10] G. Huang, R. Nammour, and W. Symes, "Full-waveform inversion via source-receiver extension," *Geophysics*, vol. 82, no. 3, R153–R171, 2017.

[11] W. W. Symes, "The search for a cycle-skipping cure: An overview," in *Institute for Pure and Applied Mathematics (IPAM): Computational Issues in Oil Field Applications*, 2017.

[12] E. Esser, L. Guasch, T. van Leeuwen, A. Y. Aravkin, and F. J. Herrmann, "Total variation regularization strategies in full-waveform inversion," *Society for Industrial and Applied Mathematics (SIAM) Journal on Imaging Sciences*, vol. 11, no. 1, pp. 376–406, 2018.

[13] L. Metivier, R. Brossier, Q. Merigot, E. Oudet, and J. Virieux, "Measuring the misfit between seismograms using an optimal transport distance: application to full waveform inversion," *Geophysical Journal International*, vol. 205, no. 1, pp. 345–377, Feb. 2016.

[14] Y. Yang and B. Engquist, "Analysis of optimal transport and related misfit functions in full-waveform inversion," *Geophysics*, vol. 83, no. 1, A7–A12, 2018.

[15] B. Peters and F. J. Herrmann, "Constraints versus penalties for edge-preserving full-waveform inversion," *The Leading Edge*, vol. 36, no. 1, pp. 94–100, 2017.

[16] B. Peters, B. R. Smithyman, and F. J. Herrmann, "Projection methods and applications for seismic nonlinear inverse problems with multiple constraints," *Geophysics*, vol. 84, no. 2, R251–R269, 2019.

[17] I. Terentyev, "A software framework for finite difference simulation," Rice University, Houston, Tech. Rep., 2009.

[18] V. Etienne, S. Operto, J. Virieux, Y. Jia, *et al.*, "Computational issues and strategies related to full waveform inversion in 3D elastic media: Methodological developments," in *80th Annual International Meeting, SEG, Expanded Abstracts*, Society of Exploration Geophysicists, 2010, pp. 1050–1054.

[19] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, "Devito: Towards a generic finite difference DSL using symbolic Python," in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, IEEE, 2016, pp. 67–75.

[20] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[21] W. W. Symes, "Reverse time migration with optimal checkpointing," *Geophysics*, vol. 72, no. 5, SM213–SM221, 2007.

[22] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[23] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *2009 International Conference on High Performance Computing Simulation*, Jun. 2009, pp. 36–43.

[24] C. Andreolli, P. Thierry, L. Borges, C. Yount, and G. Skinner, "Genetic algorithm based auto-tuning of seismic applications on multi and manycore computers," in *EAGE Workshop on High Performance Computing for Upstream*, 2014.

[25] U. Rüde, K. Willcox, L. C. McInnes, and H. D. Sterck, "Research and education in computational science and engineering," *Society for Industrial and Applied Mathematics (SIAM) Review*, vol. 60, no. 3, pp. 707–754, 2018.

[26] F. J. Herrmann, P. P. Moghaddam, and C. Stolk, "Sparsity- and continuity- promoting seismic image recovery with Curvelet frames," *Applied and Computational Harmonic Analysis*, vol. 24, pp. 150–173, 2008.

[27] F. J. Herrmann, "Randomized sampling and sparsity: Getting more information from fewer samples," *Geophysics*, vol. 75, no. 6, WB173–WB187, 2010.

[28] E. J. Candès, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006.

[29] D. Donoho, "Compressed sensing," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[30] J. Napper and P. Bientinesi, "Can cloud computing reach the Top500?" In *Proceedings of the Combined Workshops on Unconventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW '09)*, ser. UCHPC-MAW '09, Ischia, Italy: ACM, 2009, pp. 17–20, ISBN: 978-1-60558-557-4.

[31] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the Amazon Web Services cloud," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 159–168.

[32] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[33] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, Apr. 2017.

[34] P. A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G. J. Gorman, and F. J. Herrmann, "A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia," *Geophysics*, vol. 84, no. 3, F57–F71, 2019.

[35] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, *Julia: A fast dynamic language for technical computing*, `http://arxiv.org/abs/1209.5145`, Computing Research Repository (arXiv CoRR), 2012. (visited on 08/12/2016).

[36] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[37] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. A. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, *Architecture and performance of Devito, a system for automated stencil computation*, `https://arxiv.org/abs/1807.03032`, Computing Research Repository (arXiv CoRR), 2018. (visited on 07/21/2018).

[38] P. A. Witte, M. Louboutin, F. Luporini, G. J. Gorman, and F. J. Herrmann, "Compressive least-squares migration with on-the-fly Fourier transforms," *Geophysics*, vol. 84, no. 5, R655–R672, 2019.

[39] R. M. Alford, K. R. Kelly, and D. M. Boore, "Accuracy of finite-difference modeling of the acoustic wave equation," *Geophysics*, vol. 39, no. 6, pp. 834–842, 1974.

[40] M. Louboutin, P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 1: Forward modeling," *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, 2017.

[41] ——, "Full-waveform inversion, Part 2: Adjoint modeling," *The Leading Edge*, vol. 37, no. 1, pp. 69–72, 2018.

[42] T. van Leeuwen and F. J. Herrmann, "Mitigating local minima in full-waveform inversion by expanding the search space," *Geophysical Journal International*, vol. 195, pp. 661–667, Oct. 2013.

[43] G. Rizzuti, M. Louboutin, R. Wang, E. Daskalakis, and F. Herrmann, "A dual formulation for time-domain wavefield reconstruction inversion," in *89th Annual International Meeting, SEG, Expanded Abstracts*. 2019, pp. 1480–1485.

[44] A. M. Cervantes, A. Wächter, R. H. Tütüncü, and L. T. Biegler, "A reduced space interior point strategy for optimization of differential algebraic systems," *Computers and Chemical Engineering*, vol. 24, no. 1, pp. 39–51, 2000.

[45] P. Lailly and J. Bednar, "The seismic inverse problem as a sequence of before stack migrations," in *Conference on Inverse Scattering: Theory and Application*, Siam Philadelphia, PA, 1983, pp. 206–220.

[46] N. Bleistein, *Mathematical methods for wave phenomena*. Academic Press, 1984.

[47] J. L. Lions and E. Magenes, *Non-Homogeneous Boundary Value Problems and Applications*. Springer, 1972, vol. 1.

[48] G. Chavent, "Identification of functional parameters in partial differential equations," in *Joint Automatic Control Conference*, 1974, pp. 155–156.

[49] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks for Perception*, Elsevier, 1992, pp. 65–93.

[50] G. Röth and A. Tarantola, "Neural networks and inversion of seismic data," *Journal of Geophysical Research: Solid Earth*, vol. 99, no. B4, pp. 6753–6768, 1994.

[51] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[52] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. G. and Geoffrey Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[53] R. M. Weiss and J. Shragge, "Solving 3D anisotropic elastic wave equations on parallel GPU devices," *Geophysics*, vol. 78, no. 2, F7–F15, 2013.

[54] W. W. Symes and S. Dong, *The IWAVE++ inversion framework*, `http://trip.rice.edu/reports/2010/dong2.pdf`, Dec. 2010. (visited on 07/27/2018).

[55] S. Fomel, P. Sava, I. Vlad, L. Yang, and V. Bashkardin, "Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments," *Journal of Open Research Software*, vol. 1, no. 1, 2013.

[56] L. Ruthotto, E. Treister, and E. Haber, "jInv–a flexible Julia package for PDE parameter estimation," *Society for Industrial and Applied Mathematics (SIAM) Journal on Scientific Computing*, vol. 39, no. 5, S702–S722, 2017.

[57] L. A. Romero, D. C. Ghiglia, C. C. Ober, and S. A. Morton, "Phase encoding of shot records in prestack migration," *Geophysics*, vol. 65, no. 2, pp. 426–436, 2000.

[58] Y. Tang and B. Biondi, "Least-squares migration/inversion of blended data," *79th Annual International Meeting, SEG, Expanded Abstracts*, pp. 2859–2863, 2009.

[59] J. R. Krebs, J. E. Anderson, D. Hinkley, R. Neelamani, S. Lee, A. Baumstein, and M.-D. Lacasse, "Fast full-wavefield seismic inversion using encoded sources," *Geophysics*, vol. 74, no. 6, WCC177–WCC188, 2009.

[60] T. van Leeuwen, A. Y. Aravkin, and F. J. Herrmann, "Seismic Waveform Inversion by Stochastic Optimization," *International Journal of Geophysics*, no. 2011, 2011.

[61] W. Dai, W. Xin, and G. Schuster, "Least-squares migration of multisource data with a deblurring filter," *Geophysics*, vol. 76, no. 5, R135–R146, Sep. 2011.

[62] E. Haber, M. Chung, and F. Herrmann, "An effective method for parameter estimation with PDE constraints with multiple right-hand sides," *Society for Industrial and Applied Mathematics (SIAM) Journal on Optimization*, vol. 22, no. 3, pp. 739–757, 2012.

[63] E. Haber, K. Van Den Doel, and L. Horesh, "Optimal design of simultaneous source encoding," *Inverse Problems in Science and Engineering*, vol. 23, no. 5, pp. 780–797, 2015.

[64] F. J. Herrmann and X. Li, "Efficient least-squares imaging with sparsity promotion and compressive sensing," *Geophysical Prospecting*, vol. 60, pp. 696–712, 2012.

[65] X. Lu, L. Han, J. Yu, and X. Chen, "L1 norm constrained migration of blended data with the FISTA algorithm," *Journal of Geophysics and Engineering*, vol. 12, pp. 620–628, 2015.

[66] N. Tu and F. Herrmann, "Fast imaging with surface-related multiples by sparse inversion," *Geophysical Journal International*, vol. 201, no. 1, pp. 304–417, 2015.

[67] C. Li, J. Huang, Z. Li, and R. Wang, "Plane-wave least-squares reverse time migration with a preconditioned stochastic conjugate gradient method," *Geophysics*, vol. 83, no. 1, S33–S46, 2018.

[68] G. A. McMechan, "Migration by extrapolation of time-dependent boundary values," *Geophysical Prospecting*, vol. 31, no. 3, pp. 413–420, 1983.

[69] Y. A. Erlangga, C. Vuik, and C. W. Oosterlee, "On a class of preconditioners for solving the Helmholtz equation," *Applied Numerical Mathematics*, vol. 50, no. 3-4, pp. 409–425, 2004.

[70] R.-E. Plessix, "A helmholtz iterative solver for 3D seismic-imaging problems," *Geophysics*, vol. 72, no. 5, SM185–SM194, 2007.

[71] C. M. Furse, "Faster than Fourier-ultra-efficient time-to-frequency domain conversions for FDTD," in *Institute of Electrical and Electronics Engineers (IEEE) Antennas and Propagation Society International Symposium*, vol. 1, Jun. 1998, 536–539 vol.1.

[72] *Exxonmobil sets record in high-performance oil and gas reservoir computing*, https://corporate.exxonmobil.com/en/Energy-and-environment/Tools-and-processes/Exploration-technology/ExxonMobil-sets-record-in-high-performance-oil-and-gas-reservoir-computing, 2019. (visited on 05/22/2019).

[73] *Seismic processing and imaging*, https://www.pgs.com/imaging/services/processing-and-imaging/, 2019. (visited on 05/22/2019).

[74] *A close-up look at the world's largest HPC system for commercial research*, https://www.hpcwire.com/2018/01/14/close-look-worlds-largest-hpc-system-commercial-research/, 2019. (visited on 10/24/2019).

[75] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright, "Evaluating interconnect and virtualization performance for high performance computing," *Association for Computing Machinery (ACM) SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 55–60, 2012.

[76] W. Gropp, W. D. Gropp, A. D. F. E. E. Lusk, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1999, vol. 1.

[77] P. A. Witte, M. Louboutin, K. Lensink, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 3: Optimization," *The Leading Edge*, vol. 37, no. 2, 2018.

[78] P. A. Witte, M. Louboutin, and F. J. Herrmann, *Compressive least squares migration with on-the-fly Fourier transforms: Reproducible examples*, https://github.com/slimgroup/JUDI.jl/tree/master/examples/compressive_splsrtm, Apr. 2019. (visited on 10/25/2019).

[79] P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage, and F. J. Herrmann, *An event-driven approach to serverless seismic imaging in the cloud*, https://arxiv.org/abs/1909.01279, Computing Research Repository (arXiv CoRR), 2019. (visited on 09/03/2019).

[80] ——, "Event-driven workflows for large-scale seismic imaging in the cloud," in *89th Annual International Meeting, SEG, Expanded Abstracts*. 2019, pp. 3984–3988.

Part I

Software for seismic inverse problems

CHAPTER 2

A LARGE-SCALE FRAMEWORK FOR SYMBOLIC IMPLEMENTATIONS OF SEISMIC INVERSION ALGORITHMS IN JULIA

2.1 Introduction

Seismic imaging and parameter estimation are a challenging class of inverse problems, due to their large computational cost, algorithmic complexity and elaborate implementation requirements. Full-waveform inversion (FWI) [1, 2] or least-squares reverse-time migration (LS-RTM) [3, 4] involve numerical modeling of the wave equation in large two- and three-dimensional domains over many wavelengths and source locations as part of iterative algorithms and require codes that scale on large high-performance computing (HPC) clusters or on the cloud. Furthermore, seismic inverse problems are difficult to solve from a mathematical point of view as well, because they are often ill-conditioned and plagued by parasitic local minima [5].

Software packages for seismic modeling and inversion, therefore, need to meet the difficult requirement of providing both performance and abstractions for implementing complex inversion algorithms. Traditionally, production-level software frameworks in the oil and gas industry are written entirely in low-level languages such as C or Fortran, with a large amount of manual performance optimizations, while academic research frameworks such as Madagascar [6] often emphasize abstractions and reproducibility, rather than performance. As a result, the uptake of newly developed imaging and inversion algorithms by the oil and gas industry is generally slow, as it oftentimes takes programmers several months or years to incorporate new techniques into existing inversion codes. This problem is often caused by a disadvantageous structuring of the code, in which input/output (I/O) routines, wave equation solvers, parallelization and optimization algorithms are all

intermixed and difficult to modify independently. Therefore, inherently simple tasks such as swapping a line search or modifying the objective function, become complex and time-consuming operations. Furthermore, manual performance optimizations of wave equation solvers highly complicate the task of implementing correct adjoint codes at a later point in time, as would, for example, be required for least-squares migration. Finally, codes are often optimized for a specific hardware and are not portable to new platforms, making it difficult to deploy existing software to new computer architectures or the cloud.

Some of these issues are addressed in existing software packages for seismic modeling and inversion. One of the earliest seismic frameworks that introduces abstractions for prototyping wave equations for seismic modeling and inversion; thus enabling the reuse of code, is iWave++ [7, 8]. It combines a stand-alone package for solving time-domain wave equations called iWave [9], with the Rice Vector Library (RVL) [10], an object-oriented C++ library that provides abstractions for casting (seismic) inverse problems into an abstract linear algebra and optimization framework. More recent frameworks such as jInv [11] and Waveform [12], follow similar abstractions and try to overcome the trade-off between expressiveness and performance by providing an application programming interface (API) in higher-level dynamic programming languages such as MATLAB or Julia, while relying on manually optimized low-level code or libraries for solving the underlying partial differential equations (PDEs). Another trend that can be observed in both academic and industry codes, is an increase in adopting more specialized low-level languages for accelerators (graphical processing units), such as CUDA [13] and OpenCL [14]. Some of the existing seismic frameworks that fall in the broader category of (hand-tuned) modeling codes in low-level languages include the JavaSeis processing library [15], the RTM/FWI framework SAVA [16], a modeling and migration package by [17] and a finite-element inversion framework for global seismology by [18].

The seismic community is not alone with the task of writing software that is both fast and highly optimized, but at the same time provides the means for fast development of

mathematically complex algorithms. The scientific community has recently seen the rise of deep learning, a field that faces many of the same computational challenges as seismic inverse problems. Similar to FWI or LS-RTM, machine learning problems involve large data sets, complex algorithms and computationally expensive operations. In fact, we can think of a time stepping code as a feed-forward convolutional neural network and both fields use backpropagation for numerical optimization. However, in contrast to seismic inversion, uptake of new algorithms into commercial applications is extremely fast, with many of the algorithms used by major companies developed within the last months. In parts, this development is due to the wide availability of domain-specific languages (DSLs) for deep learning, such as Tensorflow [19] or PyTorch [20], used in both academia and industry. Compared to classic programming languages, DSLs offer a limited amount of functionality in exchange for domain-specific abstractions that increase productivity, while the low-level implementation details and performance optimizations are handled by computer engineers and HPC specialists. Apart from machine learning, DSLs have become popular in the field of PDEs as well, as they decouple the theoretical aspects of PDEs from the underlying, often tedious implementation of finite-difference (FD) or finite-element stencils. Two recent very popular packages for finite element modeling (FEM) that utilize DSLs are Firedrake [21] and FEniCS [22].

Based on these paradigms of domain-specific abstractions and automatic performance optimizations, we introduce a framework for seismic modeling and inversion in Julia. Borrowing ideas from machine learning frameworks and recent trends in software engineering, we develop a framework for expressing seismic PDE-constrained optimization problems like FWI and LS-RTM in terms of abstract linear algebra expressions within a high-level language, while utilizing a DSL called Devito [23, 24, 25, 26] to symbolically express the underlying PDEs and to generate fast and parallel code for solving them. Devito is a DSL embedded in Python and specifically designed for finite differences in the context of seismic modeling and inversion and offers a portable framework for automated finite-difference

code generation from PDEs. It allows the description of arbitrary time-dependent PDEs as symbolic Python expressions [27], from which optimized C code implementing a full time-stepping modeling loop is automatically generated, compiled and executed from the application environment. Here, we build upon Devito and introduce a higher-level Julia package, JUDI (JUlia Devito Inversion), to provide the means for easy development and prototyping of algorithms for seismic inverse problems on an industry scale, leading to higher productivity amongst geoscientists. As such, JUDI is the first academic seismic software framework resulting from a joint effort between geophysicists, mathematicians and HPC/compiler specialists [28], which combines advances in DSLs and compiler technologies with domain-specific requirements of geophysicists.

In the following section, we present the overall structure of JUDI and discuss its design principles and how they facilitate managing the complexity of seismic inversion frameworks. Using a series of numerical examples, we demonstrate that our approach leads to software that is highly flexible and allows implementing algorithms for FWI and LS-RTM in relatively few lines of Julia code, while providing better performance than many manually tuned codes in low-level languages.

2.2 Software structure and implementation

The Julia Devito Inversion framework is primarily designed as a research and development framework for seismic inversion, that allows us to quickly translate mathematical concepts to Julia scripts that scale to large-scale 2D and 3D problems, making it suitable for technology validation and deployment at a production level. The software is open source and available as a Julia package on Github [29]. The framework is implemented in Julia [30, 31], a high-level programming language designed for numerical computing, which offers optional typing and function overloading based on input argument types (multiple dispatch); thus providing a natural framework for abstractions. Julia also offers direct calls of Python functions without any glue code, making it convenient to interface Devito. JUDI is built

around two main applications: nonlinear inverse problems, namely full-waveform inversion, and linear least-squares problems, such as LS-RTM. The complexity of geophysical inversion frameworks arises from both the computational performance optimizations of the underlying PDE solver, required for running industry-scale problems, as well as from the need of managing extremely large amounts of data and sophisticated inversion algorithms. To break this complexity up into manageable parts, JUDI is built on the idea of multiple layers of abstractions and on keeping a clear separation between problem-dependent abstractions, parallelization and the wave equation solver (Figure 2.1). Each abstraction layer is designed to deal with one aspect of complexity:

1. Matrix-free linear operators and out-of-core data containers to address algorithmic complexity of inversion algorithms that allow users to write code that closely resembles the underlying mathematics and without having to worry about meta data, such as seismic header information.

2. A flexible high-level parallelization with built-in resilience to hardware failures that allows users to modify and adapt the parallelization to both algorithms (static or dynamic resource allocation) and the computational environment (cluster or cloud).

3. Symbolic definitions of forward and adjoint wave equations with Devito and automatic code generation to address the complexity of implementing correct forward-adjoint pairs of PDE solvers and to avoid manual performance optimizations.

Thus, the novelty of this framework is a full vertical integration of modern compiler technologies and automatic code generation into a geophysical inversion framework with problem-specific abstractions for FWI and LS-RTM in a high-level programming language, that allows researchers to use these tools both interactively during development and in batch mode for large-scale 3D problems.

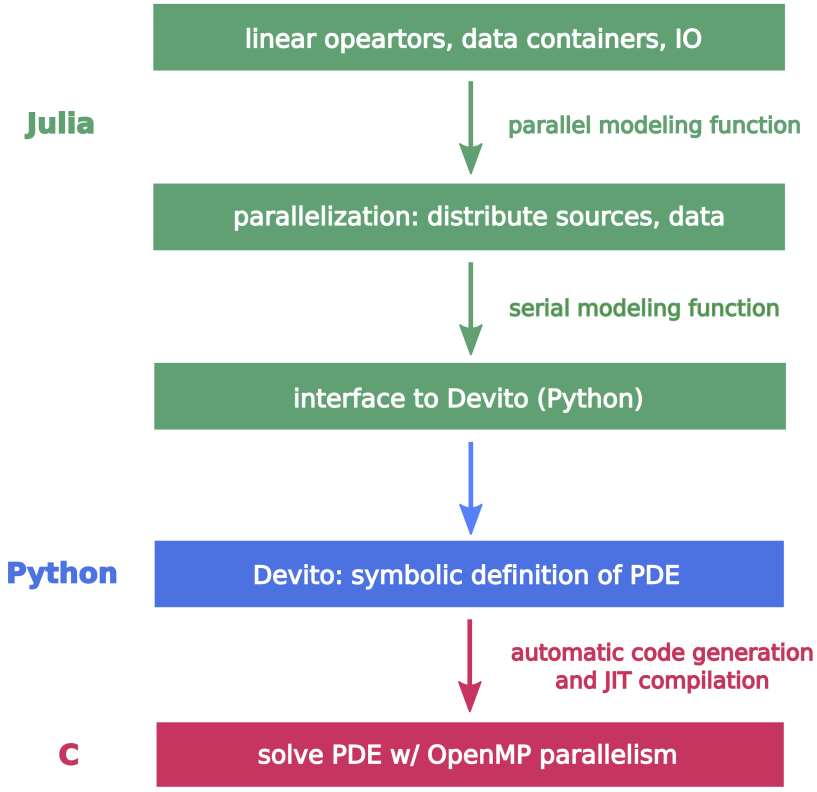


Figure 2.1: Software hierarchy of JUDI and its interface to the wave equation solver Devito. The uppermost software layer contains matrix-free operators that allow expressing PDE solvers and sampling operators as linear algebra operations. For solving multiple PDEs, data is first distributed to the available computational resources, where each worker sets up its individual PDE using Devito and generates the C code for solving it. The optimized code is compiled dynamically and called from Python.

2.2.1 Abstractions for seismic modeling and inversion

Matrix-free linear operators and vector-like seismic data containers form the first layer of JUDI, as they enable the user to express seismic modeling operations or gradients and objective functions as matrix-vector products. This provides a natural connection to linear algebra and optimization, thus making it easier for geoscientists to bridge the gap between theory and implementation. Many seismic operations like modeling/time-reverse modeling, demigration/migration or convolution/correlation can be interpreted as forward/adjoint pairs of linear operators [32]. However, rather than explicitly forming these matrices, which quickly becomes infeasible for any realistically sized problems, the actions of these matrices can be implemented as functions. Matrix-free linear operators look and behave like regular matrices, i.e., they can be multiplied with vectors or transposed, but the dense or sparse matrices are never explicitly formed and, instead, the operators contain functions that represent their actions on vectors. The concept is popular in computer science and can be found, amongst others, in PETSc [33], Trilinos [34], or Matlab libraries such as Sparco [35] and SPOT [36]. Seismic modeling and inversion frameworks that use matrix-free linear operators and data containers are RVL/iWave++ [10, 7] or the frequency-domain framework Waveform [12].

In the mathematical notation in which we express seismic inverse problems like FWI or LS-RTM, seismic data is typically denoted as a vector, while in practice, seismic data is a multi-dimensional data volume with associated meta data that contains coordinates of sources and receivers, as well as sampling rates and recording times. As pointed out in [10], mixing optimization and linear algebra algorithms with management of the dimensions and meta data of the physical observations makes codes overly complex and hard to maintain and develop. In RVL, physical data is therefore encapsulated in an abstract vector class that represents a Hilbert space on which norms and dot products are defined for a certain data type, such as seismic data. Optimization algorithms can then be implemented for these coordinate-free data types. In JUDI, we build upon this approach of RVL and iWave++

with an implementation of an abstract seismic data type called `judiVector` that looks and behaves like a regular Julia vector, but contains the seismic data in its original dimensions, together with its header information. To be able to use the data container like regular (coordinate-free) vectors, we overload common base functions and arithmetic operations for the `judiVector` type, such as size functions, norms, dot products, addition, subtraction, multiplication with scalars, transposition or concatenation. As an extension to this concept from RVL, we combine our data class with a powerful SEG-Y reader for operating on industry size out-of-core data sets. For reading and writing SEG-Y data, JUDI uses the SegyIO.jl package [37], which includes the possibility to scan large data sets of multiple terabytes and create lookup tables with a summary of SEG-Y headers and byte locations of data blocks (a data block being for example a single shot record). Blocks or shot records can then be accessed directly through their byte location within the underlying SEG-Y file, making it possible to quickly access data independently of the file size. The `judiVector` class is built around these functionalities and can be used as an out-of-core data container, in which only the lookup tables are stored in memory, rather than the full data volume, thus making it possible to work with industry-scale data sets.

Apart from the `judiVector` class for seismic data, JUDI includes matrix-free linear operators for solving (acoustic) wave equations, linearized wave equations and source/receiver projection operators. Solving a wave equation, where a seismic source `q` is injected into the subsurface and the wavefields are restricted to the receiver locations, can be expressed as the multiplication of matrices and vectors: `d=Pr*A_inv*adjoint(Ps)*q`. The operator `A_inv` denotes the inverse of the discretized wave equation for a given model (i.e.; its solution for a given right-hand side), `Ps` and `Pr` represent source/receiver projection operators and `d` is a `judiVector` containing the modeled shot record. The function `adjoint()` denotes the matrix transpose. The source/receiver projection operators are purely symbolic and do not require access to full wavefields. This means, rather than computing a full wavefield and sampling it at the receiver locations, our modeling expression

generates code with a time-stepping loop, in which the shot record `d` is generated on-the-fly, without the need to store the wavefield of the whole domain prior to restricting it to the receivers. Accessing full wavefields is generally possible by omitting the receiver projection operator, but this functionality is only viable if the necessary amount of memory to store full wavefields is available. Solutions of adjoint (time-reversed) wave equations can be obtained by transposing the modeling operator `A_inv` and by optionally restricting the solution to the source or receiver locations (Listing 2.1). Furthermore, JUDI enables users to create a linearized Born modeling operator `J` (Jacobian) from a wave equation operator, which can be used for demigration and reverse-time migration. It is important to note, that data containers and matrix-free linear operators provide only the user API for accessing data and solutions of PDEs, but are completely separated from the actual definitions of forward and adjoint wave equations themselves.

```
1 # Forward and adjoint (time-reversal) modeling
2 d_pred = Pr*A_inv*adjoint(Ps)*q
3 q_ad = Ps*adjoint(A_inv)*adjoint(Pr)*(d_pred - d_obs)
4
5 # Migration/demigration
6 J = judiJacobian(Pr*A_inv*adjoint(Ps),q) # linearize modeling ↩
     operator
7 d_lin = J*dm
8 rtm = adjoint(J)*d_lin
```

Listing 2.1: Matrix-free linear operators for nonlinear forward modeling, linearized modeling and source/receiver projections. The vector `d_pred` is a modeled seismic shot record, while `q_ad` is the solution of the corresponding adjoint wave equation, restricted to the source location. The data residual between the predicted and observed data `d_obs` acts as the adjoint source and is injected at the receiver locations, as denoted by `adjoint(Pr)`. Multiplication of the Born modeling operator `J` with a model perturbation `dm` generates a linearized shot record `d_lin`, while its adjoint migrates the data and returns an RTM image.

2.2.2 Parallelization

The matrix-free linear operators of JUDI act as wrappers around functions that contain implementations of the corresponding forward and adjoint matrix-vector products. In case of our previously introduced operators for solving wave equations (multiplications with `A_inv`) and for linearized Born modeling (multiplications with `J`), this is a function called `time_modeling`, which forms the second abstract layer of our software framework (Figure 2.1). When the modeling operators and right-hand-sides represent multiple experiments, an individual wave equation has to be solved for each source location. The `time_modeling` function therefore has a serial and a parallel function instance, meaning that there exist two functions named `time_modeling`, which only differ in how they are called (multiple dispatch) [31]. When called for more than one source/shot record, the parallel function instance of `time_modeling` is executed, which distributes the source locations and data amongst the available computational resources and then calls the serial function instance with the interface to the wave equation solver (Figure 2.1). The complete separation of the code into parallel and serial parts makes the complexity manageable and allows for easy adjustment of the parallelization to the available hardware and without having to rewrite major parts of the framework.

Julia has it own built-in parallel framework for shared and distributed memory, which is implemented in Julia itself. Parallelization is based on message-passing and features many high-level expressions that make incorporation of parallel techniques fairly simple. Generally, Julia only requires management of the master process, allowing for a clearer separation of serial and parallel code parts, since no communication statements are necessary in the serial modeling functions. For seismic modeling and inversion frameworks, the outermost parallelization is typically the distribution of sources or shots, since the objective functions often exhibit a sum structure over source locations. Solving PDEs for different source locations on multiple workers is embarrassingly parallel, since no communication is required to model wave propagation for an individual seismic experiment, as long as

the model fits on a single node, which is a reasonable assumption given current hardware configurations.

To avoid unbalanced workloads, dynamic scheduling is used to distribute the sources to the resources in the parallel pool of workers – i.e., parallel instances distributed over different computational nodes. This means source locations are assigned to workers dynamically, as they become available during execution time, which prevents resources from sitting idle. Another important feature of the Julia parallel framework is that it is relatively easy to guarantee resilience in case of hardware failures. Since large-scale seismic inverse problems involve solving a large number of PDEs (up to $10,000$ or more shot positions) as part of iterative algorithms, where programs run for several days or weeks, it is not unlikely that certain workers fail during execution time. As an alternative to saving checkpoints and restarting jobs after a crash, Julia provides functionalities for making user functions resilient to (a limited number) of hardware failures. In case of a worker exception, the PDE that was solved on that worker is resent to a different worker, while the results from the other workers remain unaffected and the program is not interrupted.

2.2.3 Interface to the wave-equation solver: Devito

The final abstraction layer of our software framework (Figure 2.1) is the serial instance of the `time_modeling` function, which contains the Julia interface to Devito [23, 25, 26]. As described in the introduction, Devito is a Python DSL for symbolic representations of PDEs, from which optimized finite-difference stencil code is automatically generated during run time and called directly from Julia. The main benefits of using Devito for solving the wave equations, rather than implementing the wave equation solvers directly in Julia itself, are significant performance improvements in speed and memory usage, as well as faster code development. The symbolic objects in Devito allow discretizing PDEs in a way that closely resembles the underlying mathematics and are completely independent of the space order of the finite-difference stencils, making it possible to experiment with different

discretizations without having to reimplement long stencil code by hand – a process that is known to be error prone. The Devito compiler transforms the symbolic specification to optimize the number of floating point operations (FLOP) as well as the memory usage, leading to fast multi-threaded C code, which performs in the range of peak performance of processors [38, 39].

This section only serves as a short summary of Devito's main features, as Devito's API and compiler are presented in separate journal articles [25, 26]. With Devito, finite-difference formulations of wave equations only need a few lines of symbolic Python. For example, the acoustic wave equation can be expressed as `pde = model.m * u.dt2 - u.laplace`, where `model` and `u` are symbolic Devito objects for velocities and wavefields. This expression can then be automatically rearranged to obtain a stencil for updating a wavefield within a time-stepping loop (see Appendix A.1 for details). Initial and boundary conditions can be specified symbolically in a similar fashion, while infinite modeling domains are simulated through absorbing boundary conditions (ABCs). A detailed walk-through of setting up forward and adjoint wave equation with Devito in the context of FWI is presented in a three-part tutorial series in the Leading Edge (TLE) [40, 41, 42]. When we want to solve a forward or adjoint wave equation, e.g., by running `d = J*dm` in Julia, C code with a time-stepping loop is automatically created from the symbolic Devito expression. The translation of the symbolic PDE representation into optimized stencil code is performed by the Devito compiler as a series of *passes*. Such passes include symbolic optimization to reduce the operation count (via the so called Devito Symbolic Engine, or DSE), loop scheduling (i.e., construction of loop nests enclosing the symbolic expressions), and loop optimization (via the Devito Loop Engine, or DLE). Thus, while the DSE captures common sub-expressions and redundant factors, i.e., it only *sees* expressions, the DLE targets the lower-level loop optimization and applies standard techniques such as blocking, as well as OpenMP parallelization [43] and vectorization.

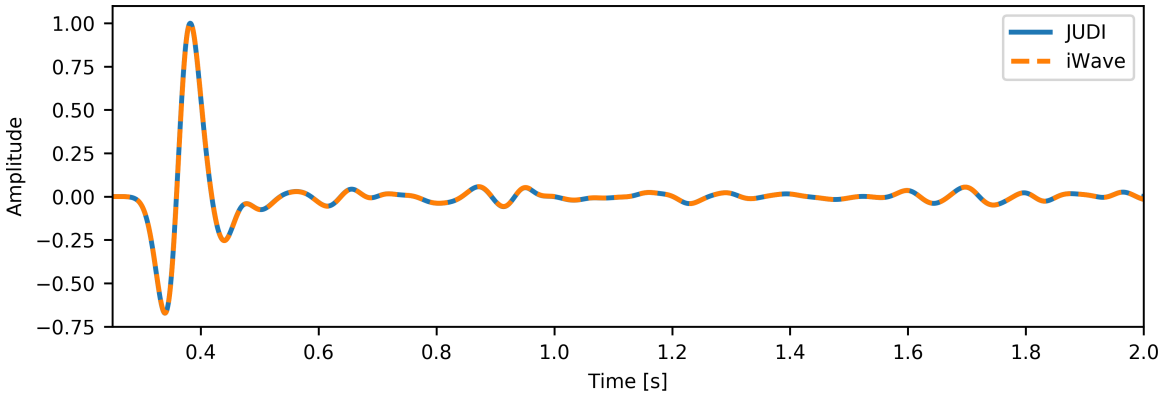
While in principle Devito allows discretizations of a large number of arbitrary PDEs,

the current release of JUDI comes with implementations of the acoustic wave equation as described in the aforementioned TLE tutorial series [40, 41]. The symbolic expressions for forward and adjoint wave equations as well as Jacobians, are defined in a separate Python module in the JUDI source code and are interfaced from Julia with the `PyCall` package [44]. The JUDI interface gathers all necessary data and modeling information from the matrix-free linear operators and interpolates source functions and shot records to the computational time axis. Arguments are passed to Python as references, avoiding data copies of wavefields; thus creating little or no memory overhead. Devito then generates optimized C code from the symbolic expressions and compiles and runs it.

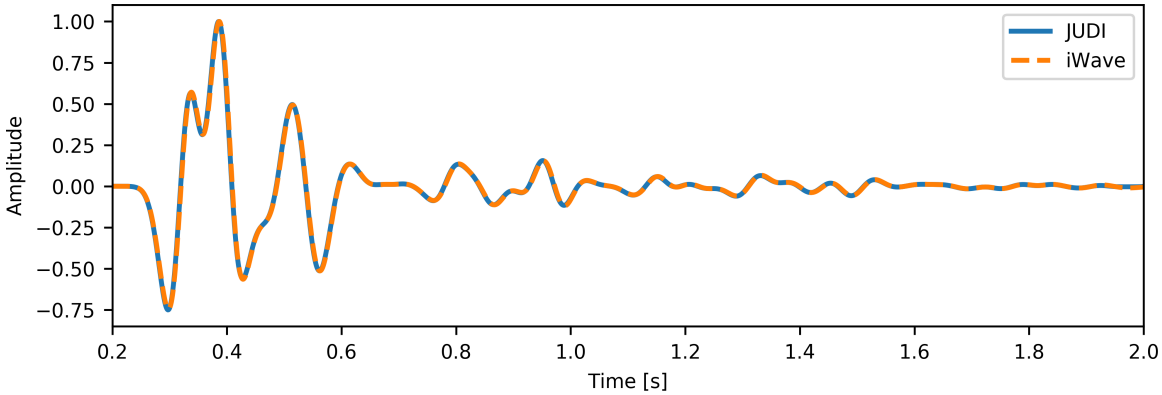
2.2.4 Unit tests

Unit testing is an essential part of any software framework, but is especially crucial for physical modeling and inversion codes that rely on a correct discretization of PDEs and accurate implementations of objective functions and gradients. Using wrongly implemented operators for linear solvers or optimization routines provides in the general case no guarantee for convergence and can potentially lead to false results [45]. For large-scale inversion codes like seismic software frameworks, code maintenance and unit testing is exceedingly challenging task, since codes initially written by geophysicists are often optimized by separate HPC experts, without careful considerations of correctly implemented physics, adjoints and gradients. With JUDI, we aim at improving code maintainability and testing through a modular code design with independent layers of abstractions, making it possible to individually test parts of JUDI, Devito and the relative interfaces.

Our first unit test validates that solving the acoustic wave equation with JUDI/Devito produces verifiably correct shot records. Since it is not possible to compute analytic solutions of the acoustic wave equation for an arbitrary velocity model, we compare modeling results of our code with an independent reference code [9, 7]. Figure 2.2 shows trace comparisons of JUDI and iWave for two different velocity models and validates that both codes



(a)



(b)

Figure 2.2: Comparison of a single traces from seismic shot records that were modeled with JUDI and iWave. Figure (a) was generated using the 2D Marmousi model and Figure (b) was modeled with the 2D Overthrust model.

create the same output. The measured error between the traces was 4 and 1 percent respectively and can be explained by differences of the spatial/temporal interpolation functions and different treatments of absorbing boundaries.

One of the fundamental unit tests for symbolic operators and functions that mimic matrix-vector and adjoint matrix-vector products, is to verify that the implementations of the operators do in fact represent correct adjoint pairs [32]. Devito itself has a unit testing framework for verifying that the implementations of forward and adjoint (linearized) wave equations are in fact representing a true adjoint pair. With the certainty that the underlying PDE solvers have correct matrix-vector and adjoint matrix-vector product implementations, the unit testing can be extended to JUDI's linear operators, namely the forward

modeling operator `M=Pr*A_inv*adjoint(Ps)` and the linearized modeling operator `J` (Listing 2.2).

```
# Adjoint test for M
err_M = dot(M*q, d) - dot(adjoint(M)*d, q)
err_M > eps && throw("Adjoint test for M failed")

# Adjoint test for J
eps1 = dot(J*dm, d_lin) - dot(adjoint(J)*d_lin, dm)
err_J > eps && throw("Adjoint test for J failed")
```

Listing 2.2: Adjoint test for JUDI's linear operators, that ensure that the modeling operators and their transposes are in fact correct forward-adjoint pairs within the computer's machine precision `eps`.

Another important test is to verify the correct implementation of our FWI gradient, which is tested by analyzing the 0th and 1st order Taylor errors of the discretization. Assuming that the FWI objective function $\Phi(\mathbf{m})$ is differentiable and smooth within the vicinity of a velocity model $\mathbf{m}$, we ensure that for a smooth reference model $\mathbf{m}_0$ and model perturbation $h \cdot \delta\mathbf{m}$, the Taylor errors (Figure 2.3) behave as predicted by Taylor's theorem for $h \to 0$:

$$\begin{aligned} &\Phi(\mathbf{m}_0 + h \cdot \delta\mathbf{m}) - \Phi(\mathbf{m}_0) = \mathcal{O}(h) \\ &\Phi(\mathbf{m}_0 + h \cdot \delta\mathbf{m}) - \Phi(\mathbf{m}_0) - h \cdot \nabla\Phi(\mathbf{m}_0)^\top \delta\mathbf{m} = \mathcal{O}(h^2). \end{aligned} \tag{2.1}$$

2.3 Numerical case studies

We will now demonstrate how our framework can be used to address various formulations of linear and nonlinear wave-equation based time-domain inverse problems. With the help of a variety of concrete examples, we underline what sets our framework apart from other seismic software frameworks:

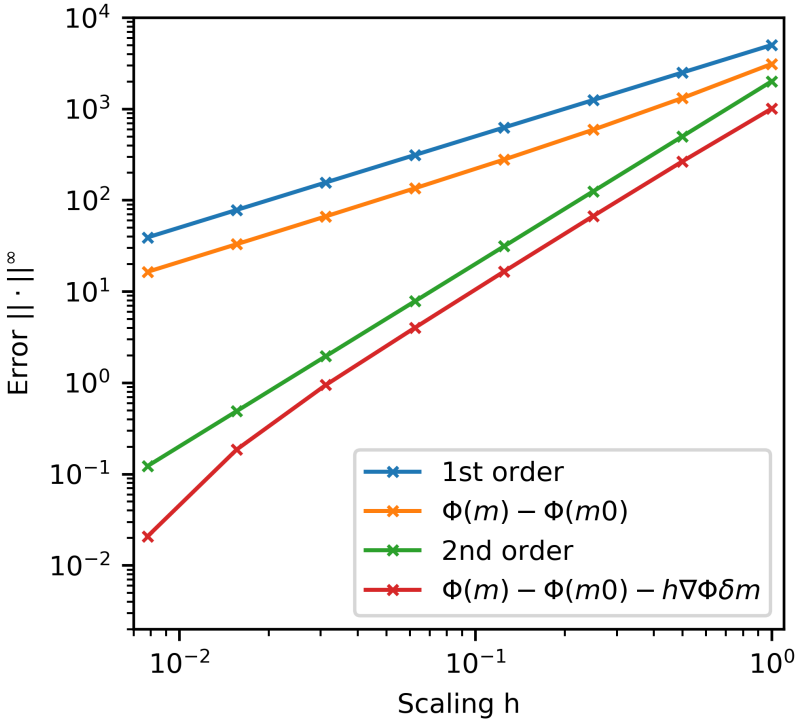


Figure 2.3: Taylor error test for the implementation of the FWI objective function and gradient. Using the gradient information causes the error to decay with 1st order as $h \to 0$, which verifies that the gradient is implemented correctly.

- the possibility to implement FWI and LS-RTM algorithms in a few lines of Julia code and to scale algorithms to large 3D problems with over 100 million unknown model parameters;
- matrix-free linear operators and out-of-core data containers that allow working with industry-sized data sets;
- full control over underlying PDEs and discretization orders through simple symbolic definitions of wave equations;
- automatic generation of highly optimized C code for solving wave equations close to processor peak performance.

We will start by showing how our software allows us to quickly implement different misfit functions for waveform inversion and how those misfit functions can be integrated into simple optimization routines or passed to sophisticated third party optimization libraries for gradient-based optimization such as minConf [46]. In the second part of this section, we address least-squares migration and demonstrate how data containers and matrix-free operators from our software framework allow us to formulate linear solvers and optimization algorithms that closely follow the underlying mathematics. To showcase the flexibility of JUDI, our examples include an implementation of a parallel optimization algorithm (elastic average stochastic gradient descent) and an implementation of LS-RTM with on-the-fly discrete Fourier transforms. We presume that the reader is familiar with the basic concepts of wave-equation based inversion and refer to [2] for a theoretical overview of FWI. Furthermore, a detailled tutorial on implementing FWI with Devito and JUDI is presented in [40] and [42].

2.3.1 Full waveform inversion

Our first numerical case study demonstrates how to implement a basic FWI example with (stochastic) gradient descent [47] and simple bound constraints on the velocity model. In

principle, JUDI allows implementing a wide range of FWI formulations, such as extended search space methods like waveform reconstruction inversion [48] or FWI via the matched source extension [49], by modifying the underlying PDEs in Python. For the sake of simplicity, we limit our examples to FWI with the adjoint state method, i.e., to objective functions of the following form:

$$\underset{\mathbf{m}}{\text{minimize}} \ \phi\Big[\mathbf{P}_r\mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^\top\mathbf{q} - \mathbf{d}_{\text{obs}}\Big], \tag{2.2}$$

where $\phi(\cdot)$ is a (smooth) misfit function and typically chosen to be the least-squares misfit $\phi = \frac{1}{2}\|\cdot\|^2$. The operators $\mathbf{P}_s$ and $\mathbf{P}_r$ denote source and receiver projections as introduced earlier and $\mathbf{A}(\mathbf{m})$ represents the discretized, time-dependent wave equation, which is a function of unknown medium parameters collected in the vector $\mathbf{m}$, such as the velocity or squared slowness. The vector $\mathbf{d}_{\text{obs}}$ represents the (vectorized) observed shot records and $\mathbf{q}$ denotes the seismic sources. The gradient of equation 2.2 with respect to the model parameters $\mathbf{m}$ is computed using the adjoint state method [1, 50], also known as a reduced space method in the optimization literature [e.g., 51] and corresponds to applying the adjoint Jacobian (migration operator) to the residual between predicted and observed data.

With JUDI, we can implement this FWI objective function as a separate Julia function called `fwi_misfit`, which takes the current model, the source and the observed data as input arguments. The function generates the predicted data for the current model and then calculates its misfit with the observed data, as well as the gradient. All necessary information for setting up the forward modeling operator and the Jacobian are entirely inferred from the input arguments. While this function is serial in itself, i.e., it can be called from the main loop of a minimization routine, the data residual and gradient are calculated in parallel, since all modeling operators are implicitly parallel. Since `fwi_misfit` is a standalone function, it can be called both within a self-implemented optimization scheme or from third-party optimization libraries, which typically require input functions of this type.

Listing 2.3 shows an example of such an objective function, in which we calculate either the standard ℓ_2-misfit $\phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|_2^2$ or the pseudo-Huber misfit $\phi(\mathbf{x}) = \epsilon^2\big(\sqrt{1+\mathbf{x}^2/\epsilon^2} - 1\big)$ [52, 53]. The vector $\mathbf{x}$ denotes the data misfit, which is in our case the difference between predicted and observed shot records and ϵ is a control parameter that determines the slope of the misfit function.

```
function fwi_misfit(model::Model, q::judiVector, d::judiVector; obj="L2")

    # Set up operators
    nt = get_computational_nt(q.geometry, d.geometry, model)
    info = Info(prod(model.n), d.nsrc, nt)
    M = judiModeling(info, model, q.geometry, d.geometry)
    J = judiJacobian(M, q)

    # Data residual, function value and gradient
    if misfit=="L2"
        r = M*q - d
        f = .5f0*norm(r)^2
        g = adjoint(J)*r
    elseif misfit=="huber"
        r = M*q - d
        f = eps^2*sqrt(1 + dot(r,r)/eps^2) - eps^2 # e.g. eps=1
        g = adjoint(J)*r/sqrt(1 + dot(r,r)/eps^2)
    end
    return f,g
end
```

Listing 2.3: Julia function for calculating the FWI ℓ_2- and pseudo-Huber misfit for a current estimate of the model, a given source `q` and observed data `d`. The matrix `M` is a combined operator, implicitly containing source/receiver projections. *Remark: The function shown here is simplified for demonstration purposes. A more efficient version without recomputing the gradient for line searches and without recomputing wavefields for the gradient is supplied in the current JUDI release.*

Setting up the FWI objective function in the specified way and using JUDI's matrix-free linear operators, has the advantage that calculating the misfit and gradient is completely decoupled from the rest of the software and can be set up independently of the optimization algorithm or the PDE solver. This means, changing the underlying wave equation to in-

clude more realistic physics or modifying the parallelization requires none (or very minor) adjustments of the functions for misfits and gradients, thus separating the set up of PDEs and optimization routines.

With our objective function in place, we can now implement a simple stochastic gradient descent algorithm in Julia (Listing 2.4). The first step in the minimization loop is to select a random subset of sources and shots, for which the gradient and objective function value will be calculated. This stochastic approach is commonly used in other fields that involve massive amounts of data and expensive evaluations of objective functions and gradients, such as training neural networks [47]. We then pass the subset of sources and data to the misfit function to calculate the gradient and objective function value for the current subset of shots. This is followed by a line search to determine the step size for updating the model. While the effectiveness of line searches for stochastic algorithms is debated by optimizers [54], empirical evidence suggests that an approximate line search can be employed successfully in applications in which only a very small number of iterations is affordable. The final step is applying the bound constraints to the velocity model to prevent velocities from becoming too small or large.

To verify that this very simple algorithm with our symbolic operators can in fact be used to successfully run FWI, we test our optimization algorithm on the 2D Overthrust model and a small data set with 97 shot records, 6 kilometers maximum offset and 3 seconds recording time. The source wavelet has a central frequency of 8 Hertz. We generate an initial model by smoothing the true model and then perform 20 iterations of the stochastic gradient descent algorithm as shown in Listing 2.4, with 20 randomly selected shots per iteration. We use bound constraints to restrict the velocity to an interval between $1,500$ and $6,500\,\mathrm{m/s}$, while keeping the water velocity fixed at $1,500\,\mathrm{m/s}$. The result after 20 iterations is shown in Figure 2.4. To make this first example easily reproducible, we use a 2D model, but our out-of-core data containers and Devito's code generation, which includes optimal checkpointing [55, 56, 57], enables us to run the same script on large-scale 3D

```
# Main loop
for j=1:maxiter

    # FWI objective function value and gradient
    i = randperm(dobs.nsrc)[1:batchsize]
    f, g = fwi_misfit(model, q[i], dobs[i])

    # line search
    step = backtracking_linesearch(vec(model0.m), g; varargs...)

    # update model and bound projection
    model.m = proj(model.m + step)

    # termination criteria
    if f <= fTerm || norm(g) <= gTerm
        break
    end
end
```

Listing 2.4: FWI with stochastic gradient descent using the previously defined `fwi_misfit` function to calculate the function value and gradient for the current subset of shots and sources. The gradient calculation is followed by a line search and a projection of the updated model onto the feasible set of velocities.

models, as we will demonstrate in the subsequent example.

As an alternative to implementing our own optimization routine, we can use the `fwi_misfit` function and interface a broad variety of Julia libraries for local gradient-based optimization, giving users access to more advanced optimization algorithms such as Quasi-Newton methods or spectral projected gradient (SPG) algorithms. For this purpose, it is typically necessary to write a small wrapper around the `fwi_misfit` function, which is customized to the individual optimization library. For our numerical example, we interface our Julia implementation of the minConf optimization library [46], which is included in our software. The library works with objective functions that take the current model estimate as the only input argument and requires that the function value and gradient are returned as a tuple. Listing 2.5 demonstrates how to wrap the `misfit_function` into an outer objective function that can be passed to minConf. Even though we hand the FWI objective function to a library over which we hold no control, the outer objective function

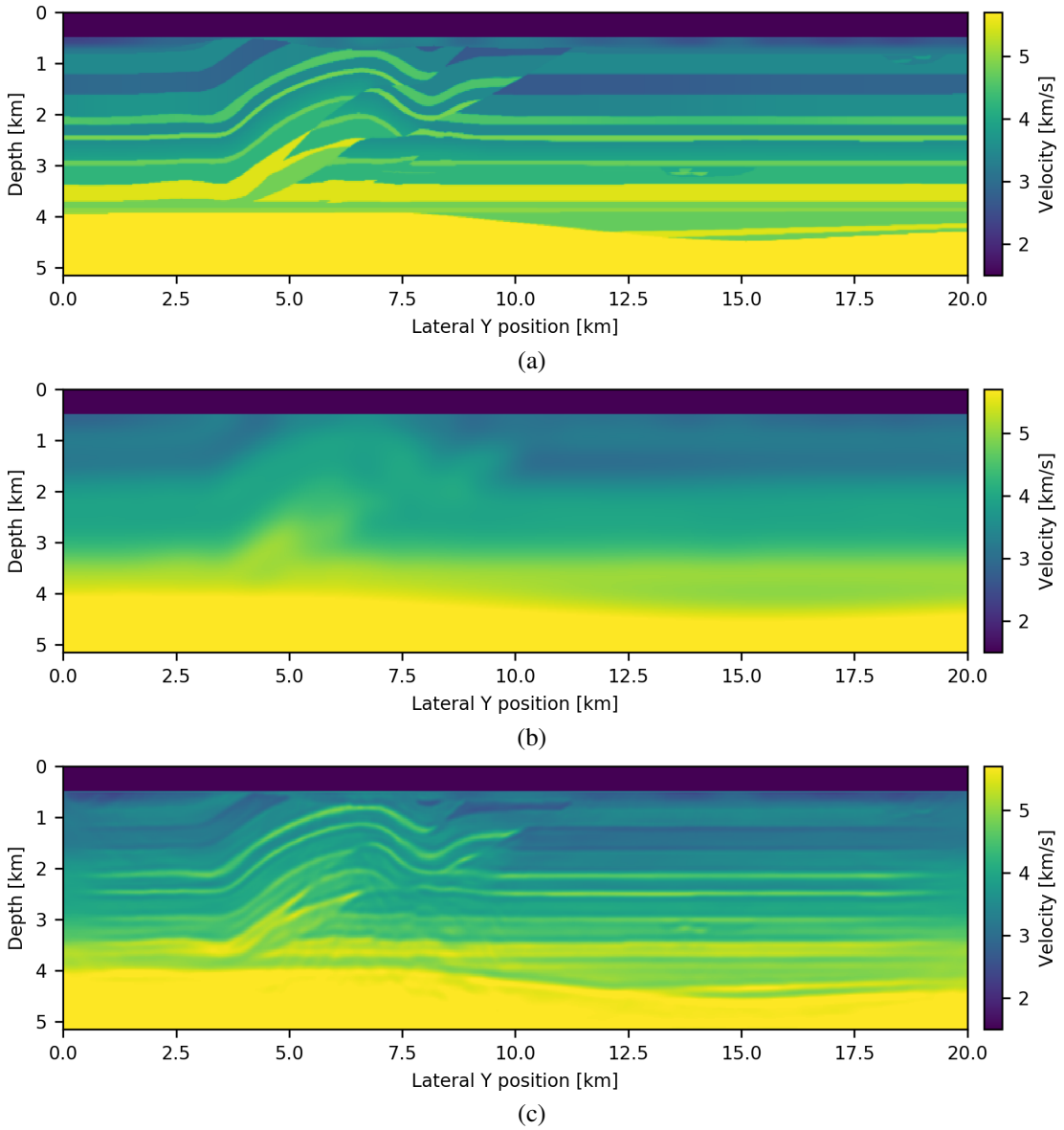


Figure 2.4: Overthrust velocity model for our 2D FWI case study (a), initial model (b) and recovered model after 20 iterations of stochastic gradient descent with bound constraints and a backtracking line search (c).

still allows us to work with randomized subsets of shots or to access and modify the gradient. In this case, we simply set the gradient in the water column to zero, but applying any type of scaling or filtering would be possible as well. We can also define additional projection operators, e.g., for enforcing sparsity, low-rank structure or monotonically increasing velocity with depth and hand them to the optimization library.

```
# optimization parameters
fevals = 15
batchsize = 1080

# objective function for minConf library
function objective_function(x)

    # set model to current estimate
    model.m = reshape(x, model.n);

    # fwi function value and gradient_test
    i = randperm(dobs.nsrc)[1:batchsize]
    f, g = fwi_misfit(model, q[i], dobs[i])

    # reset gradient in water column to 0.
    g = reshape(g, model.n); g[:, :, 1:21] .= 0f0
    return f, vec(g)
end

# FWI with spectral projected gradient
proj(x) = median([mmin x mmax], dims=2)
x, f_final = minConf_SPG(objective_function, vec(model.m), proj)
```

Listing 2.5: Wrapper around the `fwi_misfit` function for interfacing the minConf optimization library. MinConf requires objective functions with the current model vector as the only input argument and the function value and gradient as output arguments. Inside our wrapper function, we once again select a randomized subset of shots and reset the gradient in the water column to zero.

Even though the minConf library that we use in our numerical example is not primarily designed for large-scale applications, we can use it to run large-scale 3D FWI. The most computationally intensive part is evaluating the `fwi_misfit` function, which is parallelized and uses Devito to generate highly optimized C code at run time, while the optimization library in principle does not care how the objective function is evaluated. To demonstrate that our framework scales, we perform FWI on the 3D Overthrust model using the spectral projected gradient algorithm from the minConf library. Our test data set (1.2 TB) consists of over $9,400$ shot records with $8\,\mathrm{km}$ maximum offset and 3 seconds recording time and was generated with an 8 Hertz Ricker wavelet. We use the full 3D Overthrust model with a $25\,\mathrm{m}$ grid spacing, which corresponds to $801 \times 801 \times 207$ grid points and a total of over 130 million unknown parameters. Once again, we use a randomly selected subset of shots and sources in each iteration (in this case $1,080$) and we allow for a maximum number of 15 objective function evaluations. Since the forward wavefields are too large to store in memory, we enable optimal checkpointing for recomputing the wavefields during the gradient calculation [55, 56, 57]. A depth slice of the final result is shown in Figure 2.5. Apart from the minConf library, we tested interfacing the NLopt library [44], which features, amongst others, limited-memory Quasi-Newton methods.

2.3.2 Least-squares reverse time migration

The second class of seismic inverse problems that JUDI is designed for are linear least-squares problems such as LS-RTM. Like full-waveform inversion, LS-RTM is a computationally challenging problem for large-scale data sets, especially for high frequencies, and forms a broad research topic in the seismic community [e.g., 4, 58]. JUDI, with its matrix-free modeling operators and data containers, is designed to easily translate algorithms into runnable Julia code that scales to realistic models through its automatic code generation.

Once more, we will start by showing how to implement a very basic version of LS-RTM with gradient descent and then demonstrate how the code can be modified to more advanced

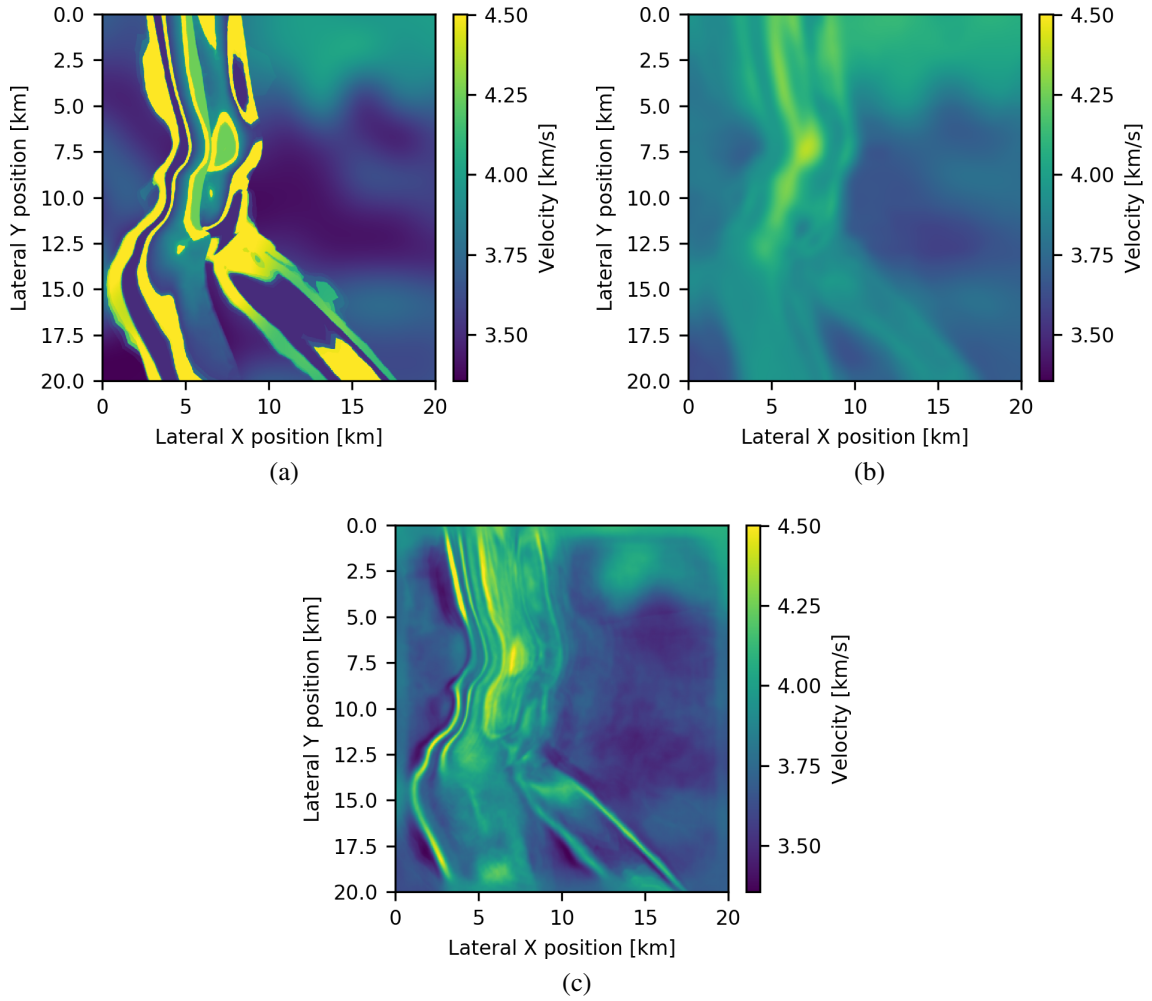


Figure 2.5: Depth slice through the original 3D Overthrust model (a), the initial model (b) and the recovered model after 15 function evaluations with minConf's spectral projected gradient algorithm (c). Some parts of the recovered model are cycle skipped, but overall minConf's SPG algorithm was able to make decent progress towards the solution. The result could be improved through a larger batch size of shots, or by adjusting the starting model.

algorithms like elastic average SGD or LS-RTM with on-the-fly Fourier transforms. For our numerical case study, we consider the standard LS-RTM objective function with left- and right-hand preconditioners $\mathbf{M}_l^{-1}$ and $\mathbf{M}_r^{-1}$, which correspond to model- and data-space preconditioners such as mutes or amplitude corrections [59]:

$$\underset{\widehat{\delta\mathbf{m}}}{\text{minimize}}\ \frac{1}{2}\|\mathbf{M}_l^{-1}\mathbf{J}\mathbf{M}_r^{-1}\widehat{\delta\mathbf{m}} - \mathbf{M}_l^{-1}\delta\mathbf{d}\|^2, \tag{2.3}$$

where $\delta\mathbf{m} = \mathbf{M}_r^{-1}\widehat{\delta\mathbf{m}}$ is the image we want to recover. As before, the matrix $\mathbf{J}$ denotes the linearized Born modeling operator and $\delta\mathbf{d}$ is the observed linearized data, i.e., shot records in which ideally all events except the reflections have been removed (such as direct and turning waves as well as surface-related multiples). The preconditioned linear least-squares problem can generally be solved with any matrix-free optimization method, while direct solvers or solvers that need access to arbitrary entries of $\mathbf{J}$ cannot be used due to the large number of dimensions and the fact that $\mathbf{J}$ is not available as an explicit matrix. The algorithm for preconditioned LS-RTM with stochastic gradient descent is given in Algorithm 2.1 and the corresponding code that implements this method using JUDI is shown in Listing 2.6. Each iteration involves selecting a random subset of shot records and extracting the corresponding blocks of rows from the demigration operator $\mathtt{J}$. The data residual and gradient are then calculated for the current subset of source locations. With preconditioners that are set up as matrix-free linear operators (using templates from the Julia operator library by [60]), the algorithm translates directly to runnable Julia code.

Algorithm 2.1 Stochastic gradient descent algorithm for least-squares RTM. The matrix $\mathbf{J}_{s(j)}$ is the subset of the demigration/migration operator that corresponds to the current subset of shots $\delta\mathbf{d}_{s(j)}$ and computes the residual and gradients in parallel. The matrices $\mathbf{M}_{l,r}^{-1}$ are left- and right-hand preconditioners in the data and model space, such as mutes, scalings or approximate inverse Hessians.

for $j = 1$ to n
  Select random subset of shot indices $s(j) \in [1...n_s]$
  $\mathbf{r}_j = \mathbf{M}_l^{-1}\mathbf{J}_{s(j)}\mathbf{M}_r^{-1}\mathbf{x}_j - \mathbf{M}_l^{-1}\delta\mathbf{d}_{s(j)}$
  $\mathbf{g}_j = \mathbf{M}_r^{-\top}\mathbf{J}_{s(j)}^{\top}\mathbf{M}_l^{-\top}\mathbf{r}_j$
  $t_j = \frac{\|\mathbf{r}_j\|^2}{\|\mathbf{g}_j\|^2}$
  $\mathbf{x}_{j+1} = \mathbf{x}_j - t_j\mathbf{g}_j$
end

The SGD algorithm in Listing 2.6 itself is serial, while the parallelization happens implicitly inside $\mathtt{J}$ in the form of distributing the source positions and data to the parallel workers. However, the flexibility of our framework allows to easily exchange the modeling parallelism for a parallel algorithm (or a combination of both). A parallel version of stochastic gradient descent is the elastic average SGD algorithm [61], as shown in Algo-

```
# Stochastic gradient descent
batchsize = 10
niter = 20

for j=1:niter

    # Compute residual and gradient
    i = randperm(d_refl.nsrc)[1:batchsize]
    r = Ml*J[i]*Mr*x - Ml*d_refl[i]
    g = adjoint(Mr)*adjoint(J[i])*adjoint(Ml)*r

    # Step size and update variable
    t = norm(r)^2/norm(g)^2
    global x -= t*g
end
```

Listing 2.6: Julia implementation of the stochastic gradient descent algorithm for LS-RTM. Our matrix-free operators for preconditioners and Jacobians allow for a direct translation of Algorithm 2.1 to runnable Julia code.

rithm 2.2. In contrast to classic SGD, the algorithm contains an additional loop over the number of parallel workers, who calculate individual gradient updates that are tied together by a center variable $\tilde{\mathbf{x}}$, which is stored and updated by the master process. Once again, this algorithm can be translated into Julia code with a moderate amount of effort (Listing 2.7). The biggest change in comparison to the SGD implementation, is a separate function that calculates the gradient and which is called in the inner loop and executed in parallel on the remote workers.

The Julia codes for serial and parallel SGD (Listings 2.6 and 2.7) are agnostic to the dimensions of the model and work for both 2D and 3D problems. Here, we show the result of running 20 iterations of EASGD on the 2D Marmousi model using 10 workers (`p=10`) and a batch size of 1. The observed data consists of 320 reflection data shot records, generated as `d_refl = J*dm`, with receivers spread out over the full model, 4 seconds recording time and 30 Hertz peak frequency. For illustration purposes and keeping the examples simple, we only demonstrate the serial and parallel implementations of stochastic gradient descent with sequential shots, but the extensions of these examples to advanced

Algorithm 2.2 Parallel version of stochastic gradient descent (elastic average SGD) for LS-RTM. Compared to the serial version, the EASGD algorithm has an additional inner loop $k = 1$ to p over the number of workers and each worker computes its individual data residual, gradient and model update $\mathbf{x}_j^k$. The master then computes the elastic average $\tilde{\mathbf{x}}_j$ from the individual model updates.

for $j = 1$ to n
  for $k = 1$ to p
    Select random subset of shot indices $s(j,k) \in [1...n_s]$
    $\mathbf{r}_j^k = \mathbf{M}_l^{-1}\mathbf{J}_{s(j,k)}\mathbf{M}_r^{-1}\mathbf{x}_j^k - \mathbf{M}_l^{-1}\delta\mathbf{d}_{s(j,k)}$
    $\mathbf{g}_j^k = \mathbf{M}_r^{-\top}\mathbf{J}_{s(j,k)}^{\top}\mathbf{M}_l^{-\top}\mathbf{r}_j^k$
    $\mathbf{x}_{j+1}^k = \mathbf{x}_j^k - \eta\mathbf{g}_j^k - \alpha\big(\mathbf{x}_j^k - \tilde{\mathbf{x}}_j\big)$
  end
$\tilde{\mathbf{x}}_{j+1} = (1-\beta)\tilde{\mathbf{x}}_j + \beta\big(\frac{1}{p}\sum_{i=1}^{p}\mathbf{x}_j^i\big)$
end

```
# Gradient function
@everywhere function update_x(Ml, J, Mr, x, d, eta, alpha, xav)
    r = Ml*J*Mr*x - Ml*d
    g = adjoint(Mr)*adjoint(J)*adjoint(Ml)*r
    return x - eta*g - alpha*(x - xav)
end
update_x_par = remote(update_x) # Parallel function wrapper

for j=1:niter
    @sync begin
        for k=1:p
            # Calculate x update in parallel
            i = randperm(d_refl.nsrc)[1:batchsize]
            xnew[:, k] = update_x_par(Ml, J[i], Mr, x[:,k],
                         d_refl[i], eta, alpha, xav)
        end
    end
    # Update average variable
    global xav = (1 - beta)*xav + beta*(1/p*sum(x, dims=2))
    global x = copy(xnew)
end
```

Listing 2.7: Implementation of the elastic average SGD algorithm for LS-RTM. Just like the algorithm, the code has an additional loop over the number of workers `p`, in which the new image is calculated by calling the remote parallel function `update_x_par` for the current subset of shots. The `@sync` statement forces the master to wait at the end of the inner loop for all workers to return their updates `xnew`. The elastic average variable `xav` is then updated by the master. The `@everywhere` statement makes the subsequent function known to all workers, not just the master process.

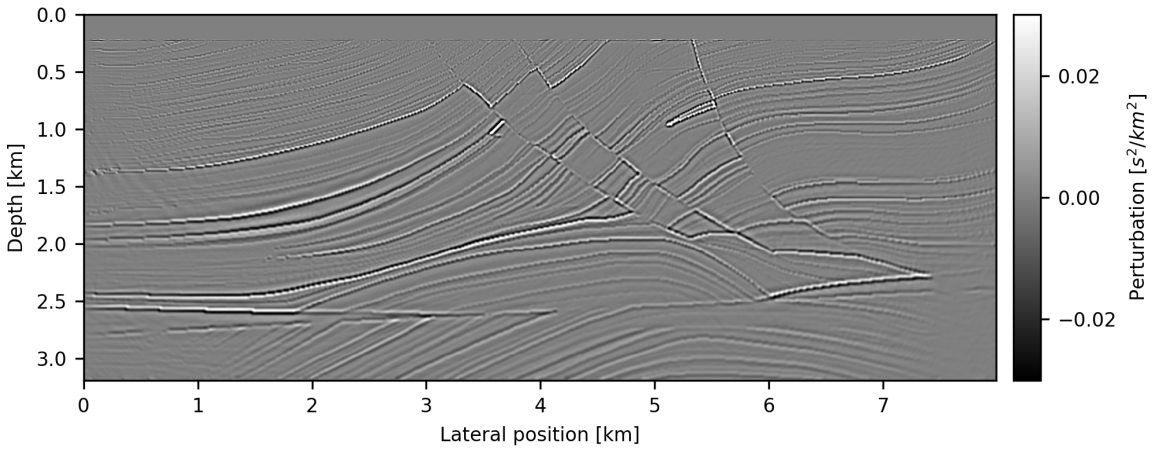


Figure 2.6: LS-RTM image of the Marmousi model after 20 iterations of the elastic average SGD algorithm. In each iteration, the 10 workers calculate their new image from single randomly selected shot and the master updates the central variable (shown here after the final iteration).

algorithms like conjugate gradient or inversion with simultaneous shots are straightforward. A demonstration of how to set up simultaneous sources with JUDI can be found in the accompanying software.

2.3.3 Compressive imaging with on-the-fly Fourier transforms

So far, all the numerical case studies shown here work with acoustic wave equations and linearized modeling operators. As discussed earlier, wave equations in our framework are set up in Python using Devito and the functions for code generation are interfaced from Julia. By modifying the Python functions that generate the underlying C code, we can implement different wave equations with density variations or anisotropy, or change imaging conditions of the migration operator. In this final example, we demonstrate how we can modify the underlying Python code for LS-RTM with on-the-fly discrete Fourier transforms (DFTs). Rather than saving the full time-domain forward wavefield for applying the zero-lag cross correlation imaging condition, we perform a real-valued DFT within the time loop and save a subset of frequency-domain wavefields (Algorithm 2.3); thus requiring substantially less memory (see [62] within the context of FWI). In the adjoint time loop for

migration (Algorithm 2.4), we perform the on-the-fly DFT on the adjoint wavefields and calculate the image by correlating the frequency-domain wavefields via simple elementwise multiplications.

Algorithm 2.3 Pseudo-code for calculating frequency-domain wavefields $\bar{\mathbf{u}}_{real}$ and $\bar{\mathbf{u}}_{imag}$ for a frequency f within the time loop of a forward modeling code. The parameter Δt is the computational time-stepping interval and nt is the total number of time steps.

for $j = 1$ to nt
  Calculate current forward wavefield: $\mathbf{u}_j = ...$
  $\bar{\mathbf{u}}_{real} = \bar{\mathbf{u}}_{real} + \mathbf{u}_j \cos\left(2\pi f j \Delta t\right)$
  $\bar{\mathbf{u}}_{imag} = \bar{\mathbf{u}}_{imag} - \mathbf{u}_j \sin\left(2\pi f j \Delta t\right)$
end

Algorithm 2.4 The frequency-domain gradient $\bar{\mathbf{g}}$ of the FWI or LS-RTM objective function is calculated by performing the on-the-fly DFT on the adjoint wavefields $\mathbf{v}_j$ and by calculating the dotwise multiplication of the real and imaginary forward and adjoint wavefields.

for $j = nt$ to 1
  Calculate current adjoint wavefield: $\mathbf{v}_j = ...$
  $\bar{\mathbf{g}} = \bar{\mathbf{g}} + 4\pi^2 f^2 \mathbf{v}_j \Big(\bar{\mathbf{u}}_{real} \cos\left(2\pi f j \Delta t\right) - \bar{\mathbf{u}}_{imag} \sin\left(2\pi f j \Delta t\right) \Big)$
end

In Python, we can use the powerful symbolic abstractions of Devito to directly translate the concept of on-the-fly Fourier transforms to Python code, from which optimized C code is generated and compiled automatically during run time. Frequency and time-domain wavefields are represented through special types (e.g., `TimeData` for wavefields) from which the time-stepping loops are constructed automatically during code generation. To implement the on-the-fly DFTs, we add the expressions shown in Listing 2.8 to our symbolic PDE expressions for forward and adjoint modeling, that are defined in the source code of JUDI.

We now repeat our numerical experiment from the previous section and perform LS-RTM on the 2D Marmousi model, using the same test data set as before. However, instead of saving the full forward wavefields in memory and calculating the gradient in the time-domain, we perform the on-the-fly DFT and only keep 10 (frequency-domain) wavefields

```
# On-the-fly real-valued DFT of forward wavefield
eqn_ufr = Eq(ufr, ufr + u*cos(2*np.pi*f*time*dt))
eqn_ufi = Eq(ufi, ufi - u*sin(2*np.pi*f*time*dt))

# On-the-fly real-valued DFT of adjoint wavefield
eqn_g = Eq(g, g+(2*np.pi*f)**2*v*(ufr*cos(2*np.pi*f*time*dt)-
                                  ufi*sin(2*np.pi*f*time*dt)))
```

Listing 2.8: On-the-fly Fourier transform for calculating frequency-domain wavefields in the forward time loop and gradients (images) in the adjoint time loop. `Eq` is a SymPy function that generates a symbolic stencil from Devito expressions and is used by Devito to automatically generate optimized C code during execution time.

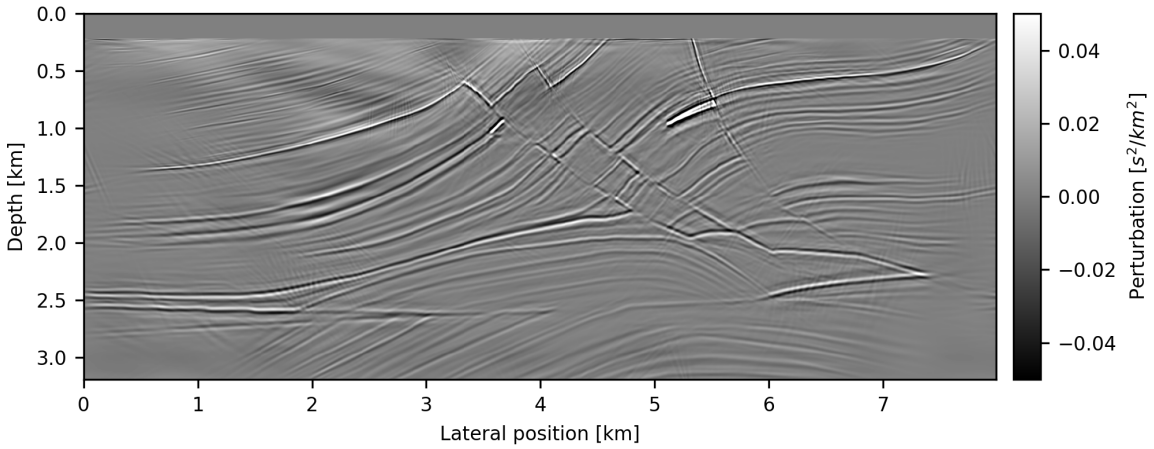


Figure 2.7: Imaging result after 32 iterations of sparsity-promoting LS-RTM with on-the-fly Fourier transforms. By only saving a few frequency-domain wavefields, this method only requires a fraction of the memory of conventional time-domain LS-RTM and therefore scales to large-scale models.

in memory from which the LS-RTM gradient is calculated. The frequencies in each iteration are selected randomly for each shot, which creates images with random noise, similar to LS-RTM with simultaneous sources or stochastic frequency-domain LS-RTM. By solving a modified version of the standard LS-RTM problem (equation 2.3) with sparsity-promotion, these artifacts can be mostly removed and we obtain an image that looks close to our previous result, but at a fraction of the memory cost (Figure 2.7). For solving the sparsity-promoting LS-RTM problem with frequency subsampling, we use the linearized Bregman method as described in [63]. The Julia code follows largely the algorithm in Listing 2.7, with an additional variable and sparsity promotion through soft thresholding.

2.4 Discussion

The numerical examples presented here are intended to demonstrate the flexibility that comes with symbolic user interfaces, making it possible to implement algorithms for wave-equation based inversion in a few lines of code and in a high-level interactive language. Our examples show that abstractions used in JUDI and Devito do not come at the cost of performance; in fact, symbolic APIs and automatic code generation are not only the key for productivity, but also the best and quickest way of obtaining efficient, functional code – code that would have taken weeks of work to optimize by hand, with no guarantees on portability and long-term maintainability. In terms of performance results for our numerical examples, we refrain from providing absolute timings, as they strongly depend on the hardware, amount of available computational resources and parameters, such as the stencil order. A more meaningful metric for performance measurements is the roofline model [38, 64, 39], which measures usage of the hardware compared to the best performance that can theoretically be achieved for a given discretization and implementation. A roofline analysis of Devito is provided in [25] and [26], with Devito reaching up to 60 percent of maximum achievable performance, depending on the stencil order, which is significantly higher than the average performance of finite-difference stencil codes.

With JUDI, we introduce a seismic modeling and inversion framework based on domain-specific abstractions and automatic code generation, which combines components in different languages (Julia, Python, C) into a single package. This stands in contrast to a more traditional approach to high-performance computing in low-level programming languages and with manual performance optimizations. JUDI provides abstractions for definitions of objective functions and optimization algorithms in Julia, an interface to Python for symbolic definitions of forward and adjoint wave equations, while optimized time-stepping code for solving PDEs is automatically generated by the Devito compiler. Exposing Devito's capabilites through JUDI's abstract linear algebra operators, provides

researchers with the means to implement modern optimization algorithms on a high abstraction level and without having to implement low-level stencil codes. This structure makes it possbile to independently modify each aspect of seismic inverse problems, such as changing the definition of wave equations, without having to modify the optimization algorithm or implementing a new misfit function without having to worry about the parallelization. Exposing the symbolic interfaces in high-level languages such as Julia and Python makes the software usable by a wide range of users, not just experienced C or Fortran programmers.

This approach to scientific computing is strongly inspired by recent machine learning frameworks such as Tensorflow or PyTorch, which make building blocks of deep learning tools available to a wide audience and therefore promote the fast progress of this field. With packages like Tensorflow, any interested researcher can implement and train a neural network in a few hours, e.g. by following simple online tutorials, without having to know how to implement convolutions on graphical processing units. With JUDI, we apply this paradigm to seismic inverse problems and introduce a software framework that makes it possible to build workflows and algorithms for FWI and LS-RTM on a high abstraction level and without requiring the knowledge of how to implement finite-difference time-stepping codes in C. This approach also simplifies the implementation of adjoint wave equations and verifiably correct gradients – tasks that are often impossible to accomplish in reasonable amounts of time when working with hand-tuned codes in low-level languages. Some disadvantages that come with JUDI and this approach to software design, are the additional amount of work that comes with the interaction of different packages or programming languages. Furthermore, this type of code development requires a stronger interaction between geophysicists and software engineers/compiler specialists, since inversion codes typically require very problem-specific functionalities, such as source/receiver interpolations. However, we believe that the advantages greatly outweigh these downsides and pay off in the long run.

2.5 Conclusion

As seismic inversion problems become increasingly mathematically and computationally complex, geophysicists need to rethink the paradigms for developing software packages. Adapting manually optimized codes in low-level languages to new hardware environments such as the cloud or implementing sophisticated algorithms for inversion is often impossible to accomplish in reasonable amounts of time. One of the core problems amounts to the fact that algorithms, parallelization and performance optimizations are oftentimes interwoven and become impossible to modify independently. With the Julia Devito Inversion framework, we introduce an open-source software package that aims at overcoming these issues through independent layers of abstractions that break the complexity into manageable parts. We neither argue that JUDI is the only possible way of implementing these principles, nor that Julia is the only viable programming language for this, or that one specific language is superior to another. Rather, we hope to stimulate a discussion on how to engineer seismic and geophysical software in a way that helps progressing the field and making it more accessible and user-friendly to our community. With the framework introduced in this work, we aim to promote software based on symbolic user interfaces and automatic code generation, rather than manually optimized inversion codes in low-level languages. We demonstrate that abstractions and performance are not mutually exclusive, but that symbolic interfaces can greatly facilitate the implementation of seismic inversion algorithms. Based on experiences from the related machine learning community, we believe that moving to a new paradigm for geophysical software can only happen with close interactions and collaborations between academia and industry, but that a shift towards mutually developed open-source software will eventually benefit both sides, as it is a prerequisite for driving innovations.

REFERENCES

[1] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, no. 8, p. 1259, 1984.

[2] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, WCC127–WCC152, Nov. 2009.

[3] T. Nemeth, C. Wu, and G. T. Schuster, "Least-squares migration of incomplete reflection data," *Geophysics*, vol. 64, no. 1, pp. 208–221, 1999.

[4] Y. Tang and B. Biondi, "Least-squares migration/inversion of blended data," *79th Annual International Meeting, SEG, Expanded Abstracts*, pp. 2859–2863, 2009.

[5] W. W. Symes, "The search for a cycle-skipping cure: An overview," in *Institute for Pure and Applied Mathematics (IPAM): Computational Issues in Oil Field Applications*, 2017.

[6] S. Fomel, P. Sava, I. Vlad, L. Yang, and V. Bashkardin, "Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments," *Journal of Open Research Software*, vol. 1, no. 1, 2013.

[7] W. W. Symes and S. Dong, *The IWAVE++ inversion framework*, `http://trip.rice.edu/reports/2010/dong2.pdf`, Dec. 2010. (visited on 07/27/2018).

[8] W. W. Symes, D. Sun, and M. Enriquez, "From modelling to inversion: Designing a well-adapted simulator," *Geophysical Prospecting*, vol. 59, no. 5, pp. 814–833, 2011.

[9] I. Terentyev, "A software framework for finite difference simulation," Rice University, Houston, Tech. Rep., 2009.

[10] A. D. Padula, S. D. Scott, and W. W. Symes, "A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 36, no. 2, 8:1–8:36, Apr. 2009.

[11] L. Ruthotto, E. Treister, and E. Haber, "jInv–a flexible Julia package for PDE parameter estimation," *Society for Industrial and Applied Mathematics (SIAM) Journal on Scientific Computing*, vol. 39, no. 5, S702–S722, 2017.

[12] C. D. Silva and F. J. Herrmann, "A unified 2D/3D large-scale software environment for nonlinear inverse problems," *Association for Computing Machinery (ACM) Transactions on Mathematical Software (TOMS)*, vol. 45, 7:1–7:35, 2017.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[14] J. Tompson and K. Schlachter, *Introduction to the OpenCL programming model*, `https://cims.nyu.edu/~schlacht/OpenCLModel.pdf`, 2012. (visited on 11/14/2016).

[15] S. Hassanzadeh and C. C. Mosher, "Javaseis: Web delivery of seismic processing services," pp. 2055–2057, 1997.

[16] D. Koehn, *Sava*, `https://github.com/daniel-koehn/SAVA`, 2017. (visited on 07/27/2018).

[17] J. Thorbecke, *Opensource*, `https://github.com/JanThorbecke/Open Source`, 2017. (visited on 07/27/2018).

[18] L. Krischer, A. Fichtner, S. Zukauskaite, and H. Igel, "Large-scale seismic inversion framework," *Seismological Research Letters*, vol. 86, no. 4, p. 1198, 2015.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. G. and Geoffrey Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[21] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *Association for Computing Machinery (ACM) Transactions on Mathematical Software (TOMS)*, vol. 43, no. 3, p. 24, 2017.

[22] A. Logg, K.-A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method, The FEniCS Book*, ser. Lecture Notes in Computational Science and Engineering. Springer, 2012, vol. 84, ISBN: 978-3-642-23098-1.

[23] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, "Devito: Towards a generic finite difference DSL us-

ing symbolic Python," in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, IEEE, 2016, pp. 67–75.

[24] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman, "Devito: Automated fast finite difference computation," in *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, IEEE, 2016, pp. 11–19.

[25] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[26] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. A. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, *Architecture and performance of Devito, a system for automated stencil computation*, `https://arxiv.org/abs/1807.03032`, Computing Research Repository (arXiv CoRR), 2018. (visited on 07/21/2018).

[27] D. Joyner, O. Certik, A. Meurer, and B. E. Granger, "Open source computer algebra systems: SymPy," *Association for Computing Machinery (ACM) Communications in Computer Algebra*, vol. 45, no. 3/4, pp. 225–234, Jan. 2012.

[28] F. Luporini, D. A. Ham, and P. H. Kelly, "An algorithm for the optimization of finite element integration loops," *Association for Computing Machinery (ACM) Transactions on Mathematical Software (TOMS)*, vol. 44, no. 1, p. 3, 2017.

[29] P. A. Witte, M. Louboutin, and F. J. Herrmann, *The Julia Devito Inversion Framework (JUDI)*, `https://github.com/slimgroup/JUDI.jl`, Apr. 2019. (visited on 04/15/2019).

[30] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, *Julia: A fast dynamic language for technical computing*, `http://arxiv.org/abs/1209.5145`, Computing Research Repository (arXiv CoRR), 2012. (visited on 08/12/2016).

[31] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *Society for Industrial and Applied Mathematics (SIAM) Review*, vol. 59, no. 1, pp. 65–98, 2017.

[32] J. Claerbout, *Earth Soundings Analysis: Processing Versus Inversion*. Blackwell Scientific Publications, 1992, ISBN: 9780865422100.

[33] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijhkout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, L. Curfman McInnes,

K. Rupp, B. Smith, S. Zampini, S. Zhang, and H. Zhang, *PETSc users manual*, 3.7, Argonne National Laboratory, 2016.

[34] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[35] E. van den Berg, M. P. Friedlander, G. Hennenfent, F. J. Herrmann, R. Saab, and Ö. ". Yilmaz, "Algorithm 890: Sparco: A testing framework for sparse reconstruction," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 35, no. 4, pp. 1–16, Feb. 2009.

[36] E. van den Berg and M. P. Friedlander, *Spot - A linear operator toolbox*, `https://github.com/mpf/spot`, 2013. (visited on 07/27/2018).

[37] K. Lensink, *SegyIO.jl*, `https://github.com/slimgroup/SegyIO.jl`, 2017. (visited on 11/07/2019).

[38] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the Association for Computing Machinery (ACM)*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[39] M. Louboutin, M. Lange, F. J. Herrmann, N. Kukreja, and G. Gorman, "Performance prediction of finite-difference solvers for different computer architectures," *Computers and Geosciences*, vol. 105, pp. 148–157, 2017.

[40] M. Louboutin, P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 1: Forward modeling," *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, 2017.

[41] ——, "Full-waveform inversion, Part 2: Adjoint modeling," *The Leading Edge*, vol. 37, no. 1, pp. 69–72, 2018.

[42] P. A. Witte, M. Louboutin, K. Lensink, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 3: Optimization," *The Leading Edge*, vol. 37, no. 2, 2018.

[43] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *Institute of Electrical and Electronics Engineers (IEEE) Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[44] S. Johnson, *Calling Python functions from the Julia language*, `https://github.com/JuliaPy/PyCall.jl`, 2017. (visited on 07/27/2018).

[45] G. L. Zeng and G. T. Gullberg, "Unmatched projector/backprojector pairs in an iterative reconstruction algorithm," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Medical Imaging*, vol. 19, no. 5, pp. 548–555, May 2000.

[46] M. Schmidt, E. van den Berg, M. Friedlander, and K. Murphy, "Optimizing costly functions with simple constraints: A limited-memory projected Quasi-Newton algorithm," in *Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS) 2009*, D. van Dyk and M. Welling, Eds., vol. 5, Clearwater Beach, Florida, Apr. 2009, pp. 456–463.

[47] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of Computational Statistics 2010: 19th International Conference on Computational Statistics Paris France*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[48] T. van Leeuwen and F. J. Herrmann, "Mitigating local minima in full-waveform inversion by expanding the search space," *Geophysical Journal International*, vol. 195, pp. 661–667, Oct. 2013.

[49] G. Huang, R. Nammour, and W. Symes, "Full-waveform inversion via source-receiver extension," *Geophysics*, vol. 82, no. 3, R153–R171, 2017.

[50] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[51] A. M. Cervantes, A. Wächter, R. H. Tütüncü, and L. T. Biegler, "A reduced space interior point strategy for optimization of differential algebraic systems," *Computers and Chemical Engineering*, vol. 24, no. 1, pp. 39–51, 2000.

[52] A. Guitton and W. W. Symes, "Robust inversion of seismic data using the Huber norm," *Geophysics*, vol. 68, no. 4, pp. 1310–1319, 2003.

[53] T. van Leeuwen, A. Y. Aravkin, H. Calandra, and F. J. Herrmann, "In which domain should we measure the misfit for robust full waveform inversion?" In *75th Conference and Exhibition, EAGE, Expanded Abstracts*, Jun. 2013.

[54] C. Tan, S. Ma, Y.-H. Dai, and Y. Qian, "Barzilai-borwein step size for stochastic gradient descent," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 685–693.

[55] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *As-*

sociation for Computing Machinery (ACM) Transactions on Mathematical Software, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[56] W. W. Symes, "Reverse time migration with optimal checkpointing," *Geophysics*, vol. 72, no. 5, SM213–SM221, 2007.

[57] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, *High-level Python abstractions for optimal checkpointing in inversion problems*, `https://arxiv.org/abs/1802.02474`, Computing Research Repository (arXiv CoRR), Jan. 2018. (visited on 02/14/2018).

[58] S. Dong, J. Cai, M. Guo, S. Suh, Z. Zhang, B. Wang, and Z. Li, "Least-squares reverse time migration: Towards true amplitude imaging and improving the resolution," in *82nd Annual International Meeting, SEG, Expanded Abstracts*, 2012, pp. 1–5.

[59] F. J. Herrmann, P. P. Moghaddam, and C. Stolk, "Sparsity- and continuity- promoting seismic image recovery with Curvelet frames," *Applied and Computational Harmonic Analysis*, vol. 24, pp. 150–173, 2008.

[60] H. Modzelewski, *JOLI - Julia Operator Library*, `https://github.com/slimgroup/JOLI.jl`, 2017. (visited on 07/27/2018).

[61] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," in *Advances in Neural Information Processing Systems*, 2015, pp. 685–693.

[62] L. Sirgue, J. Etgen, U. Albertin, and S. Brandsberg-Dahl, *System and method for 3D frequency domain waveform inversion based on 3D time-domain forward modeling*, US Patent 7,725,266, May 2010.

[63] F. J. Herrmann, N. Tu, and E. Esser, "Fast "online" migration with Compressive Sensing," *77th Conference and Exhibition, EAGE, Expanded Abstracts*, 2015.

[64] C. Andreolli, P. Thierry, L. Borges, C. Yount, and G. Skinner, "Genetic algorithm based auto-tuning of seismic applications on multi and manycore computers," in *EAGE Workshop on High Performance Computing for Upstream*, 2014.

Part II

Compressive seismic imaging

CHAPTER 3

COMPRESSIVE LEAST-SQUARES MIGRATION WITH ON-THE-FLY FOURIER TRANSFORMS

3.1 Introduction

Reverse-time migration (RTM) is an increasingly popular wave-equation based seismic imaging algorithm that corresponds to applying the adjoint of the Born scattering operator to observed reflection data [1, 2]. Without extensive preconditioning, applying the adjoint operator leads to an image with incorrect amplitudes, imprints of the source wavelet, finite apertures and therefore blurred reflectors. To overcome these issues and invert the Born scattering operator, imaging can be formulated as a linear least-squares optimization problem, in which the mismatch between observed and modeled data is minimized in a least-squares sense. Least-squares migration was introduced for ray-based imaging methods first [3, 4, 5] and later extended to wave-equation based imaging (LS-RTM) [6, 7, 8, 9].

The successful deployment of LS-RTM in practice is currently hampered by two distinct computational challenges. First of all, conventional LS-RTM requires the migration/demigration of all shot records in each iteration of gradient-based optimization algorithms, making this approach prohibitively expensive for large-scale data sets with thousands of individual shot records. To save computational resources, shots can therefore be subsampled or combined into supergathers/simultaneous shots, which avoids having to treat every shot separately in each iteration [7, 10, 11, 12, 13, 14, 15]. The resulting LS-RTM formulations can then be solved using stochastic optimization methods, such as stochastic gradient descent or variants of the stochastic conjugate gradient method [e.g. 16, 17]. Since the migration of simultaneous shot records leads to cross-talk in the seismic image,

the resulting artifacts need be addressed by additional constraints such as smoothing constraints [18] or transform-domain sparsity [19, 11, 20, 21]. Common transforms that lead to sparsity of seismic images include the wavelet, seislet or curvelet transforms [22, 23, 24].

The second challenge of LS-RTM are the large requirements of data movement and fast memory access for computing the gradient of the objective function with the adjoint-state method [25, 26]. The gradient for one shot record is computed by solving an adjoint wave equation, with the data residual between the predicted and observed data as the adjoint source, and requires access to the forward wavefields in reverse order. The forward wavefields are obtained by forward propagating the seismic source for the respective shot record, but they are typically too large to store in memory. To overcome this issue, a common strategy is to either write compressed wavefields to disk, or to store only a small subset of uncompressed wavefields, while the in-between wavefields are recomputed from checkpoints [27, 28] or the model boundary [29]. These approaches therefore offer a possible trade-off between memory usage and computational cost, as more wavefields need to be recomputed for a smaller number of checkpoints. Further alternatives for circumventing the storage of time-domain wavefields are discussed in [30]. Storing or recomputing wavefields is especially expensive for seismic imaging, since it is typically carried out at higher frequencies than full-waveform inversion (FWI) and thus involves substantially larger wavefields and models due to small grid spacings.

An alternative to recomputing time-domain wavefields from checkpoints, is to use time-to-frequency conversions to extract monochromatic frequency-domain wavefields from a time-stepping loop and to compute gradients for a small subset of frequencies. This approach circumvents the problem of having to store or recompute wavefields for a large number of time steps, as frequency-domain gradients are computed individually for one frequency at a time. Modeling frequency-domain wavefields with a time-domain modeling code is a well established approach in scientific computing and several algorithms

exist to perform time-to-frequency conversions, including discrete on-the-fly Fourier transforms (DFTs) or linear equation methods [refer to 31, for an overview]. A conversion method in the context of seismic modeling using phase-sensitive detections (PSDs) is presented in [32]. Extracting single frequency-domain wavefields from a time-stepping loop is even possible without any conversion at all and can be obtained by simply propagating a monochromatic source function to a steady state [33].

Time-to-frequency conversion methods enjoy great popularity in the context of full-waveform inversion [34, 35, 36, 37, 38], as it is generally desirable to carry out the inversion for single or few frequencies at a time [39, 40]. For seismic imaging on the other hand, the goal is to obtain a high-definition image with a broad frequency band, making it typically necessary to compute gradients for a large-number of evenly-spaced frequencies. Hence, we present a workflow for least-squares RTM using small subsets of randomly selected frequencies and shot records, with sparsity promotion to address the subsampling related imaging artifacts. In contrast to earlier works by [19] and [41], we use on-the-fly Fourier tranforms to compute gradients in the frequency domain with a highly optimized time-domain modeling code [42, 43]. On-the-fly DFTs not only allow us to compute an arbitrary number of frequencies in a single time-stepping loop, but also to scale the inversion to high frequencies, without solving large-scale 2D, and in particular 3D, Helmholtz equations. In our numerical examples, we demonstrate that compressive imaging with sparsity promotion (SPLS-RTM) and on-the-fly DFTs yields images of similar quality as time-domain LS-RTM, but without having to store or recompute time-domain wavefields and with a significantly reduced number of wave equation solves, using as few as two passes through the data. In the discussion, we analyze the asymptotic behaviour of memory requirements and computational cost for imaging with on-the-fly DFTs and compare it to optimal checkpointing. Thus, the contribution of this work is the formulation of frequency domain LS-RTM with a time-domain modeling operator and on-the-fly Fourier transforms using shot and frequency subsampling to overcome the prohibitively high cost

of least squares RTM. The shot and frequency-subsampling effectively turn LS-RTM into an underdetermined compressed sensing problem, with computational flexibility regarding batch sizes and number of iterations, which can be customized according to the available computational resources. Furthermore, this paper introduces a forward-adjoint pair for imaging the impedance in both the time and frequency domain (with or without on-the-fly DFTs) and presents a quantitative and qualitative comparison of time-domain LS-RTM and frequency-domain LS-RTM with on-the-fly DFTs.

3.2 Theory and methodology

3.2.1 Frequency-domain least-squares migration with time-domain modeling

To circumvent the problem of having to store time-domain wavefields for computing the adjoint-state gradient, we formulate the least-squares reverse-time migration objective function in the frequency domain, using the frequency-domain linearized Born scattering operator $\mathbf{J}$:

$$\underset{\delta\mathbf{m}}{\text{minimize}} \quad \sum_{j=1}^{n_s}\sum_{k=1}^{n_f}\frac{1}{2}\left\|\mathbf{J}(\mathbf{m}_0,\bar{q}_{jk})\;\delta\mathbf{m}-\bar{\mathbf{d}}_{jk}^{\text{obs}}\right\|_2^2. \tag{3.1}$$

The vector $\mathbf{m}_0$ denotes the vectorized migration velocity model in squared slowness and $\delta\mathbf{m}$ is the unknown model perturbation (i.e. the seismic image). The vector $\bar{\mathbf{d}}_{jk}^{\text{obs}}$ is the observed reflection data in the frequency-domain (denoted by bars) of the j^{th} source location and the k^{th} frequency and $\bar{q}_{jk}$ is the complex-valued monochromatic source. The objective function is computed as the sum over all n_s source positions and n_f temporal frequencies. To model the predicted linearized data in the Fourier domain, we have to compute the action of the linearized Born modeling operator on the current image, $\bar{\mathbf{d}}_{jk}^{\text{pred}}=\mathbf{J}(\mathbf{m}_0,\bar{q}_{jk})\;\delta\mathbf{m}$, which corresponds to solving:

$$\bar{\mathbf{d}}_{jk}^{\text{pred}}=-\mathbf{P}_r\mathbf{H}(\mathbf{m}_0)^{-1}\text{diag}\left[\frac{\partial\mathbf{H}(\mathbf{m}_0)}{\partial\mathbf{m}}\mathbf{H}(\mathbf{m}_0)^{-1}\mathbf{p}_s^*\bar{q}_{jk}\right]\delta\mathbf{m}, \tag{3.2}$$

where $\mathbf{H}(\mathbf{m}_0)$ is the frequency-domain modeling operator and $\mathbf{P}_r$ is a projection operator that restricts the wavefield to the receiver locations. Accordingly, the vector $\mathbf{p}_s^*$ is the source injection operator, which is a column vector of zeros with value one at the source location. The asterisk denotes the adjoint of complex vectors and matrices, also known as the Hermitian adjoint.

While formulating the LS-RTM objective in the frequency domain avoids storing the history of the time-domain wavefields, it in principle requires inverting the Helmholtz matrix for modeling observed data and computing gradients. To avoid inverting large-scale Helmholtz systems, which are known to be ill-conditioned for large-scale systems with high frequencies, we instead compute the frequency-domain data with a time-domain modeling code [31, 32, 34]. This combines the best of both worlds, as we can use a highly optimized time-domain modeling code for the PDE solves, while computing gradients memory efficiently in the frequency domain, where gradients are separable over frequencies. This means it is possible to compute the gradient for single frequencies at a time, as an element-wise product of the corresponding forward and adjoint monochromatic wavefields. In principal, modeling the frequency response of the linearized modeling operator involves computing the inverse DFT of the source wavelet $\bar{q}_{jk}$, solving the linearized wave equation in the time domain and then performing a time-to-frequency conversion of the single scattered wavefields for all frequencies (i.e. using a DFT). This is followed by extracting the k^{th} frequency through a frequency restriction operator $\mathbf{R}_k$ and restricting the wavefield to the receiver locations (through a receiver projection operator $\mathbf{P}_r$). Overall, we have:

$$\bar{\mathbf{d}}_{jk}^{\text{pred}} = -\mathbf{P}_r\mathbf{R}_k\mathbf{F}\mathbf{A}(\mathbf{m}_0)^{-1}\text{diag}\left[\frac{\partial\mathbf{A}(\mathbf{m}_0)}{\partial\mathbf{m}}\mathbf{A}(\mathbf{m}_0)^{-1}\mathbf{F}^*\mathbf{R}_k^*\mathbf{p}_s^*\bar{q}_{jk}\right]\delta\mathbf{m}, \tag{3.3}$$

where $\mathbf{A}(\mathbf{m})$ is the discretized time-domain wave equation and $\mathbf{F}$ is the discrete Fourier matrix. As mentioned, $\mathbf{R}_k$ is a restriction operator that extracts the k^{th} frequency of the wavefield, while its adjoint zero-padds a frequency wavefield along the frequency axis, so

that we can compute its inverse temporal Fourier transform. However, in the actual implementation of equation 3.3, we combine the time-to-frequency conversion and the extraction of one or multiple frequencies into a single step, by performing on-the-fly DFTs for the respective frequencies, instead of an explicit DFT of all frequencies. Accordingly, we never explicitly zero-padd the source wavelet to perform an inverse DFT, but simply inject the time-domain wavelet.

To obtain the gradient of the frequency-domain LS-RTM objective function, we have to compute the action of the complex conjugate linearized modelig operator on the data residual, i.e.; $\bar{\mathbf{g}}_{jk} = \mathbf{J}(\mathbf{m}_0, \bar{q}_{jk})^*(\bar{\mathbf{d}}^{\text{pred}}_{jk} - \bar{\mathbf{d}}^{\text{obs}}_{jk})$. The expression for the gradient can be derived by taking the conjugate transpose of equation 3.2 and boils down to calculating the pointwise product of the (complex) forward and adjoint wavefields $\bar{\mathbf{u}}_{jk}$ and $\bar{\mathbf{v}}_{jk}$ [44]:

$$\bar{\mathbf{g}}_{jk} = -\text{Re}\Big[\text{diag}\big(\omega_k^2 \bar{\mathbf{u}}_{jk}\big)^* \bar{\mathbf{v}}_{jk}\Big]. \tag{3.4}$$

The scalar ω_k^2 is the squared angular frequency $\omega_k = 2\pi f_k$ and Re denotes the real part of the gradient. Once again, we do not compute the forward and adjoint frequency-domain wavefields by inverting the Helmholtz equation, but by solving time-domain wave equations, followed by time-to-frequency conversions for the respective frequencies. In an analogous manner to equation 3.3, we obtain the forward wavefield $\bar{\mathbf{u}}_{jk}$ for the j^{th} source location and k^{th} frequency by solving:

$$\bar{\mathbf{u}}_{jk} = \mathbf{R}_k \mathbf{F} \mathbf{A}(\mathbf{m}_0)^{-1} \mathbf{F}^* \mathbf{R}_k^* \mathbf{p}_s^* \bar{q}_{jk}. \tag{3.5}$$

The adjoint wavefield is obtained accordingly, by solving an adjoint time-domain wave equation with the data residual as the adjoint source. In general, this is different from time reversing the data residual and using the forward modeling operator, as it involves inverting

the correct adjoint of the forward modeling operator:

$$\bar{\mathbf{v}}_{jk} = \mathbf{R}_k \mathbf{F} \mathbf{A}(\mathbf{m}_0)^{-*} \mathbf{F}^* \mathbf{R}_k^* \mathbf{P}_r^* (\bar{\mathbf{d}}_{jk}^{\text{pred}} - \bar{\mathbf{d}}_{jk}^{\text{obs}}), \tag{3.6}$$

with $\mathbf{A}(\mathbf{m}_0)^{-*}$ being the solution of the adjoint time-domain wave equation. In summary, we have derived an expression for computing the predicted linearized data in the frequency-domain using a time-domain modeling code, as well as corresponding expressions for the gradient of the LS-RTM objective function. Thus, these quantities can be computed with highly-optimized time-domain modeling codes instead of Helmholtz solvers, whose success heavily rely on the underlying preconditioners and linear solvers. However, the analytical expressions in this section contain explicit Fourier transforms of the time-domain wavefields that require access to their full time history in memory, which is what we wanted to avoid in the first place. Luckily, we can avoid having to explicitly compute discrete Fourier transforms of the time-domain wavefields, by computing the DFTs *on the fly*.

3.2.2 Computing on-the-fly Fourier transforms

To avoid saving the full time-dependent wavefield in memory and taking its Fourier transform after modeling, we compute Fourier domain wavefields for a given frequency f_k during the forward or reverse time loops on the fly. In the context of full-waveform inversion, this approach has been introduced by [34] and has since then appeared in a number of publications related to FWI [e.g. 35, 36, 37, 38]. The on-the-fly Fourier transforms correspond to computing the frequency-domain wavefields as a running sum over the current time-domain wavefield $\mathbf{u}_i$ of the i^{th} time step within a time modeling loop, multiplied with a complex exponential [31]. To simplify the implementation and avoid complex arrays, we individually compute the real and imaginary part of the frequency domain wavefields. The

on-the-fly Fourier transform of the forward wavefields is then given by:

$$\begin{aligned}\bar{\mathbf{u}}_{jk}^{\text{real}} &= \sum_{i=1}^{n_t} \cos(2\pi f_k i\Delta t)\mathbf{u}_i, \\ \bar{\mathbf{u}}_{jk}^{\text{imag}} &= -\sum_{i=1}^{n_t} \sin(2\pi f_k i\Delta t)\mathbf{u}_i.\end{aligned} \tag{3.7}$$

This expression can be fairly easily incorporated into an existing time-domain modeling code and only involves initializing the two frequency-domain wavefields with zeros and adding the current time-domain wavefield multiplied with the sine and cosine terms during each time step. To obtain the linearized data in the frequency-domain (equation 3.3), we technically have to perform the on-the-fly DFT on the linearized (single-scattered) wavefields, rather than on the source wavefields. Alternatively, it is possible to model time-domain shot records and perform the frequency conversion after modeling, since time-domain shot records themselves are not as big as time-domain wavefields and can generally be stored in memory. In the numerical examples section, we demonstrate, that this step is actually not necessary for LS-RTM, as we can simply use full time-domain sources and shot records (data residuals) as forward and adjoint sources. This means, we only use on-the-fly DFTs to compute wavefields for the gradient in the frequency-domain, but we work with time-domain data and sources.

The adjoint frequency-domain wavefields (equation 3.6) that are necessary for computing the LS-RTM gradient, are computed in the same manner as the forward wavefields, namely by performing an on-the-fly DFT on the adjoint time-domain wavefields $\mathbf{v}_i$. These wavefields are computed by solving an adjoint (i.e. time-reversed) wave equation $\mathbf{A}^{-\star}$, just as in conventional RTM. The LS-RTM gradient itself can be computed in the same time loop as the adjoint frequency-domain wavefields. We replace the complex Fourier domain wavefield $\bar{\mathbf{u}}_{jk}$ in equation 3.4 with the real and imaginary parts as given by equation 3.7 and compute the real part of the gradient through an on-the-fly DFT of the adjoint time-domain

wavefield $\mathbf{v}_i$:

$$\bar{\mathbf{g}}_{jk} = -\sum_{i=1}^{n_t} (2\pi f_k)^2 \text{diag}\Big[\bar{\mathbf{u}}_{jk}^{\text{real}} \cos(2\pi f_k i\Delta t) - \bar{\mathbf{u}}_{jk}^{\text{imag}} \sin(2\pi f_k i\Delta t)\Big]\mathbf{v}_i. \tag{3.8}$$

This equation gives us an expression for calculating the LS-RTM gradient in the frequency domain with a time-modeling code for any given frequency f_k, not just evenly spaced frequencies as obtained with an FFT. Unlike in the time domain or in equations 3.4 – 3.6, we never need to store the full time-domain wavefield $\mathbf{u}_i$ with $i = 1, \ldots, n_t$ in history. Instead, forward wavefields and gradients are computed as a running sum within forward or adjoint time loops and only require the storage of two wavefields per frequency at a time (real and imaginary parts). During a single time-stepping loop, it is of course possible to compute multiple monochromatic wavefields for different frequencies by creating an inner loop within the on-the-fly DFT over the number of frequency-domain wavefields. While this does not increase the number of time-stepping loops (PDE solves), every additional frequency increases both memory requirements and computational cost within the respective time loop.

3.2.3 A forward-adjoint pair for imaging the impedance

The gradient of the LS-RTM objective function that we derived in the previous sections uses the zero-lag cross-correlation imaging condition and maps seismic reflections in the observed data to a perturbation in the medium parameters, which are in this case the velocity in squared slowness (s^{-2} km^2). One of the well known shortcomings of imaging velocity perturbations with the zero-lag cross-correlation imaging condition, are low frequency imaging artifacts that result from backscattering of the source wavefield [e.g. 45, 46]. This issue is especially problematic for imaging salt bodies, as high velocity contrasts in the migration velocity model lead to reflections/backscattering of the down-going wavefield, thus creating strong low-frequency artifacts in the image (Figure 3.1). This phenomenon

has been well studied using radiation pattern analysis [47, 48] and can be addressed by directional filtering of the wavefields, reparametrizations of the image or alternative imaging conditions. Here, we follow the approach from [49] and address this issue by deriving the gradient of the LS-RTM objective function using the linearized inverse scattering imaging condition (ISIC). The image/gradient with ISIC is given by the sum of two terms, in which the low frequency artifacts have opposite signs and cancel each other, while the reflectors have equal signs and stack coherently [50]. In the frequency domain, the imaging condition is defined as [49]:

$$\bar{\mathbf{g}}_{jk} = -\mathrm{Re}\left[\mathrm{diag}\big(\omega_k^2 \bar{\mathbf{u}}_{jk}\big)^* \mathrm{diag}\big(\mathbf{m}_0\big)\bar{\mathbf{v}}_{jk} - \sum_{l=1}^{n_{dim}} \mathrm{diag}\left(\frac{\partial \bar{\mathbf{u}}_{jk}}{\partial \mathbf{x}_l}\right)^* \frac{\partial \bar{\mathbf{v}}_{jk}}{\partial \mathbf{x}_l}\right], \tag{3.9}$$

where n_{dim} is the number of spatial dimensions and $\frac{\partial}{\partial \mathbf{x}_i}$ is the first spatial derivative of the respective dimension. The first term of this equation corresponds to the cross-correlation imaging condition as defined by equation 3.4 with an additional pointwise multiplication with the background squared slowness vector $\mathbf{m}_0$, while the second term is the sum of pointwise products of the first spatial derivatives of forward and adjoint wavefields. In Appendix A.2, we show that the linearized inverse scattering imaging condition corresponds in fact to imaging the acoustic impedance, making the gradient $\bar{\mathbf{g}}_{jk}$ an impedance update, rather than a velocity perturbation (squared slowness) update. Since we want to use ISIC in the context of imaging with on-the-fly Fourier transforms, we follow the approach from the previous section and compute the gradient in a (reverse) time-stepping loop, in which the adjoint frequency-domain wavefields $\bar{\mathbf{v}}_{jk}$ are obtained through an on-the-fly Fourier transform of the adjoint time-domain wavefield $\mathbf{v}_i$:

$$\begin{aligned}\bar{\mathbf{g}}_{jk} = -\sum_{i=1}^{n_t} \Bigg\{ &(2\pi f_k)^2 \mathrm{diag}\Big[\bar{\mathbf{u}}_{jk}^{\text{real}} \cos(2\pi f_k i\Delta t) - \bar{\mathbf{u}}_{jk}^{\text{imag}} \sin(2\pi f_k i\Delta t)\Big]\mathrm{diag}\big(\mathbf{m}_0\big)\mathbf{v}_i - \\ &\sum_{l=1}^{n_{dim}} \mathrm{diag}\left[\frac{\partial \bar{\mathbf{u}}_{jk}^{\text{real}}}{\partial \mathbf{x}_l}\cos(2\pi f_k i\Delta t) - \frac{\partial \bar{\mathbf{u}}_{jk}^{\text{imag}}}{\partial \mathbf{x}_l}\sin(2\pi f_k i\Delta t)\right]\frac{\partial \mathbf{v}_i}{\partial \mathbf{x}_l}\Bigg\}.\end{aligned} \tag{3.10}$$

As before, the frequency domain source wavefields $\bar{\mathbf{u}}_{jk}$ are computed in a separate forward time-stepping loop with an on-the-fly DFT of the forward time-domain wavefield (equation 3.7). As discussed earlier, we can think of the gradient expression as the action of an adjoint linear operator $\mathbf{J}^*$ on the LS-RTM data residual, which maps a perturbation in the data to a perturbation in the model (the seismic image). To use ISIC/impedance imaging in the context of LS-RTM, we need to derive the corresponding forward operator, i.e.; the linear map from the image domain to the data domain [51]. Once again, we model the linearized data with a time-stepping modeling code, followed either by an on-the-fly DFT of the linearized wavefield or a time-to-frequency conversion of the time-domain shot record. First, we compute the perturbed (single scattered) wavefield $\delta\mathbf{u}_i$ of the i^{th} time step, such that the expression is the (time-domain) adjoint operation of equation 3.9:

$$\delta\mathbf{u}_i = -\mathbf{A}(\mathbf{m}_0)^{-1}\left\{\text{diag}\left(\frac{\partial^2\mathbf{u}_i}{\partial t^2}\right)\text{diag}\big(\mathbf{m}_0\big)\delta\mathbf{z} - \sum_{l=1}^{n_{dim}}\frac{\partial}{\partial\mathbf{x}_l}\left[\text{diag}\left(\frac{\partial\mathbf{u}_i}{\partial\mathbf{x}_l}\right)\delta\mathbf{z}\right]\right\}, \quad (3.11)$$

where the vector $\delta\mathbf{z}$ denotes the impedance. The real and imaginary parts of the linearized data are then obtained by performing the on-the-fly DFT on the scattered time-domain wavefields $\delta\mathbf{u}_i$ and by restricting the wavefield to the receiver locations:

$$\begin{aligned}\bar{\mathbf{d}}_{jk}^{\text{pred}_r} &= \mathbf{P}_r\sum_{i=1}^{n_t}\cos(2\pi f_k i\Delta t)\delta\mathbf{u}_i,\\ \bar{\mathbf{d}}_{jk}^{\text{pred}_i} &= -\mathbf{P}_r\sum_{i=1}^{n_t}\sin(2\pi f_k i\Delta t)\delta\mathbf{u}_i.\end{aligned} \quad (3.12)$$

To obtain linearized data for an impedance image $\delta\mathbf{z}$ in the time domain, we can simply omit the on-the-fly DFT and directly apply the receiver restriction operator $\mathbf{P}_r$ to the time-domain wavefield $\delta\mathbf{u}_i$. This allows us to use the modeling operator in equation 3.11 also for conventional time-domain LS-RTM with impedance imaging. The optimization algorithm for LS-RTM with on-the-fly DFTs and sparsity-promotion that is described in the in the following section, is independent of which imaging condition is used and works for imaging

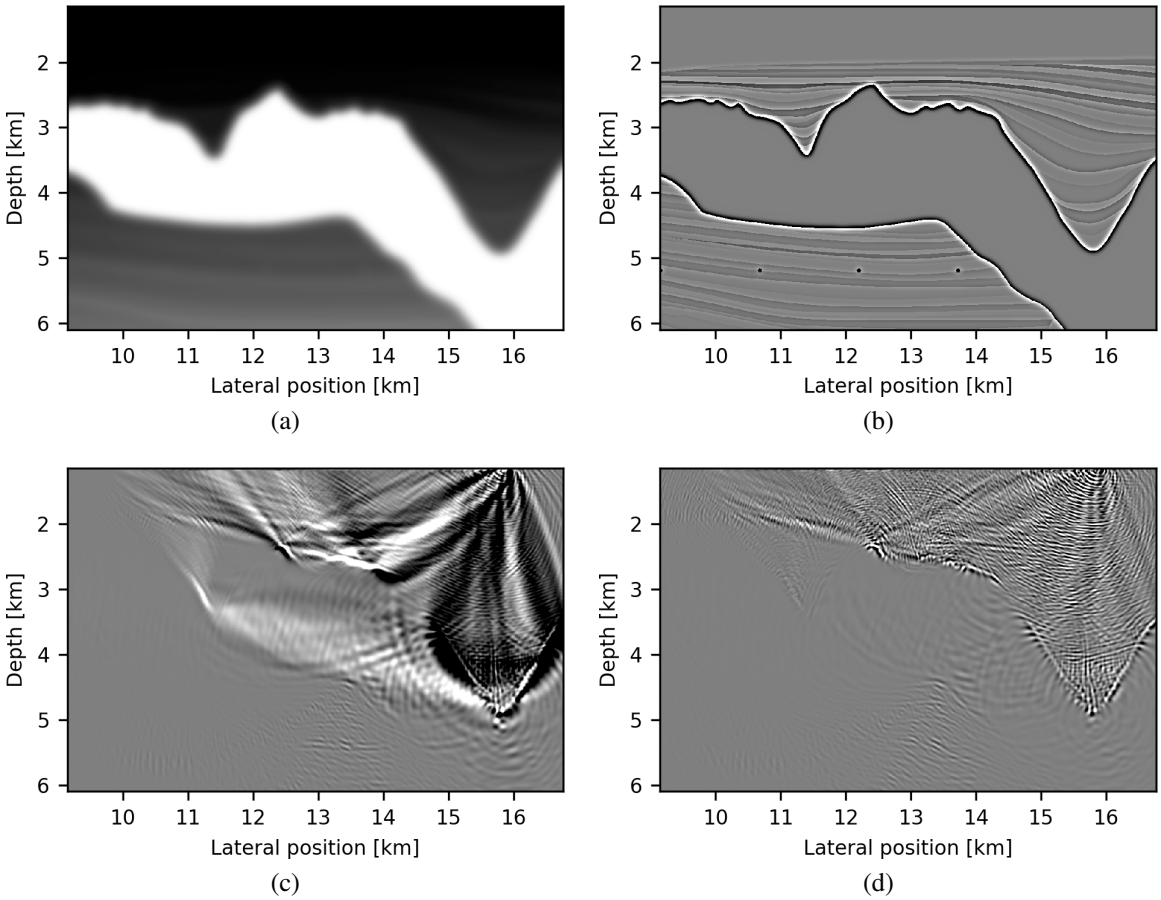


Figure 3.1: Comparison of RTM with the zero-lag cross-correlation imaging condition (c) versus the linearized inverse scattering imaging condition (d). The top row shows the smooth migration velocity model (a) and the true image (b). Both results were computed for a single shot record using 20 randomly chosen frequencies, which expectedly leads to crosstalk (spectral leakage) in the image and one-sided illumination of the salt body. However, the migration result with the cross-correlation imaging condition also suffers from strong low-frequency backscattering artifacts, whereas the inverse-scattering imaging condition is able to successfully suppress this energy.

the impedance using equation 3.10, as well as imaging velocity contrasts (equation 3.8). In theory, the equations provided here for acoustic/impedance modeling and computing the gradients using on-the-fly DFTs are exact adjoints and are able to pass adjoint tests if implemented as described.

3.2.4 Sparsity-promoting least-squares migration

The expressions we derived in the last sections allow us in principle to perform frequency-domain LS-RTM using a time-modeling code and without having to store time histories of wavefields. However, for conventional LS-RTM, the gradient is given by the full sum of all frequencies and source locations. The number of frequencies is determined by the recording length of the shot records and their sampling ratio (i.e. by the corresponding Nyquist frequency) and is generally quite large. Performing on-the-fly DFTs for a large number of frequencies in a single time loop is not only computationally expensive, but also requires the storage of all those wavefields and therefore defeats the purpose of this approach. Our method is therefore most useful, when we compute the gradients of the LS-RTM objective function for a small subset of frequencies, rather than for all frequencies. I.e.; for a single source index j, we compute $\hat{n}_f \ll n_f$ frequencies within one time loop, which requires the storage of $2\hat{n}_f$ wavefields for the gradient.

In the context of FWI, computing the gradient for a subset of frequencies makes sense, as it is generally desirable to invert the velocity model from low to high frequencies, using single or few temporal frequencies at a time [39, 40]. For seismic imaging on the other hand, the goal is to obtain a high resolution image from data with a broad frequency spectrum. As pointed out in [52], using all temporal frequencies for seismic imaging is generally not necessary, as frequencies are typically oversampled and the sampling ratio depends on the scattering angles. This allows us to image seismic data using subsets of evenly spaced frequencies, where the frequency interval is determined by the recording length, which in turn depends on the target depth and the overburden velocity [53]. Alternatively, the field of compressed sensing (CS) has recently provided a theoretical framework for sampling signals far below the Nyquist criterion using *randomized sampling* [54, 55]. The core idea of CS is to break the coherency of subsampling artifacts (aliases) into incoherent noise, by sampling on a non-uniform grid and to recover the signal using denoising techniques. Applied to LS-RTM, compressed sensing translates to working with

random subsets of shots and frequencies, rather than evenly spaced samples. As is well known from compressive seismic imaging in the frequency domain [19, 56, 21], migrating data that consists of subsets of randomly selected frequencies leads to noise/crosstalk in the images, similar to artifacts from simultaneous shots with source encoding [e.g. 57, 7]. This is demonstrated in Figure 3.2, which compares migration results in the time and frequency domain for single and multiple shots and using frequency subsampling. While migrating a single shot record using 20 randomly selected frequencies leads to an image with seemingly strong coherent artifacts, these artifacts convert to random noise after stacking 10 migrated shots, where each shot is migrated with a different set of randomly selected frequencies. Selecting the frequencies randomly is crucial for being able to recover the true image with sparsity-promoting minimization, as it breaks the coherency of the subsampling artifacts (namely aliases). Subsampling frequencies periodically, as well as truncating the ends of the frequency spectrum, leads to coherent artifacts (aliases), which are more difficult to separate from the signal.

Due to the fact that the frequency subsampling artifacts appear as incoherent noise in the image, it is possible to apply post-migration denoising techniques [refer to 58, for an overview] or to address the artifacts as part of the inversion process itself and by modifying the least-squares RTM objective function. We follow the approaches in [23] and [41] and formulate LS-RTM as a sparsity-promoting minimization problem of the following form:

$$\begin{aligned} &\underset{\delta\mathbf{z}}{\text{minimize}} \quad \lambda||\mathbf{C}\,\delta\mathbf{z}||_1 + \frac{1}{2}||\mathbf{C}\,\delta\mathbf{z}||_2^2 \\ &\text{subject to:} \sum_{j=1}^{n_s}\sum_{k=1}^{n_f} \left\| \mathbf{M}_l^{-1}\mathbf{J}(\mathbf{m}_0,\bar{q}_{jk})\mathbf{M}_r^{-1}\delta\mathbf{z} - \mathbf{M}_l^{-1}\bar{\mathbf{d}}_{jk}^{\text{obs}} \right\|_2 \leq \sigma. \end{aligned} \tag{3.13}$$

The goal of this problem is to minimize the combined ℓ_1-ℓ_2-norm of the unknown parameters, which are in our case the curvelet coefficients of the acoustic impedance $\delta\mathbf{z}$, obtained through multiplication with the forward curvelet transform $\mathbf{C}$. This is subject to the constraint that the predicted linearized data, given by the action of the linearized modeling

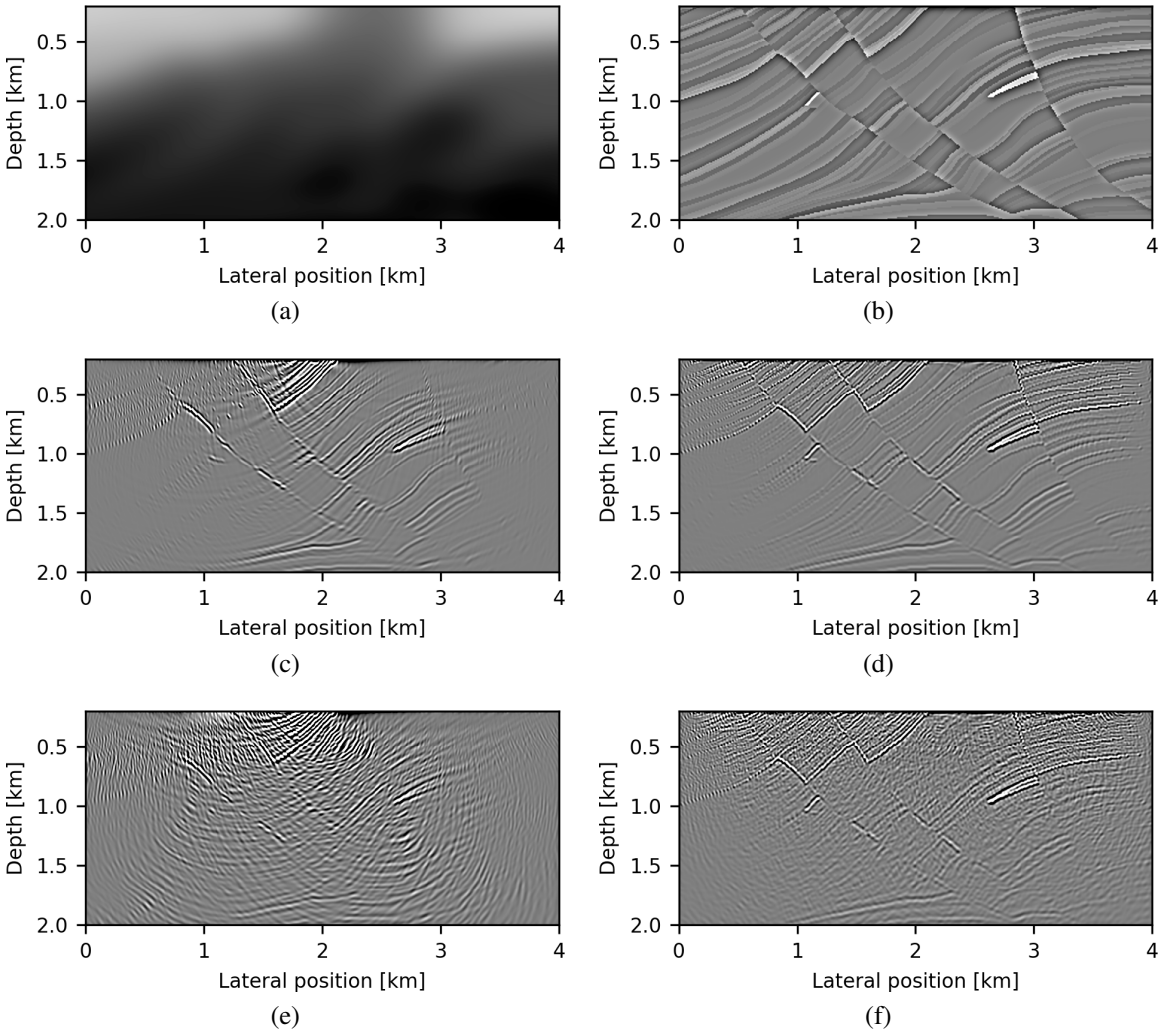


Figure 3.2: Comparisons of reverse-time migration in the time and frequency domain with on-the-fly Fourier transforms and frequency subsampling. Figure (a) is the migration velocity model and (b) is the true image. Figures (c) and (d) are the results of migrating a single shot and 10 shots in the time domain. Figures (e) and (f) are the corresponding results in the frequency domain with a subset of 20 randomly selected frequencies per shot. The migrated shot record in the frequency domain for 20 randomly selected frequencies (e) shows a very weak signal-to-noise ratio in comparison to its time-domain equivalent (c). However, when stacking the migration results of 10 shots, where each migrated shot consists of a different set of randomly selected frequencies, the reflectors stack coherently, while the subsampling artifacts appear as incoherent noise (f).

operator $\mathbf{J}$ on $\delta\mathbf{z}$, fits the observed reflections $\bar{\mathbf{d}}_{jk}^{\text{obs}}$ within some noise level σ. The matrices $\mathbf{M}_l^{-1}$ and $\mathbf{M}_r^{-1}$ are left- and right-hand preconditioners intended to improve the condition number of the system, such as mutes, depth scalings or half integrations [23]. In the nu-

merical examples, we use image mutes to set the water column to zero, as well as depth scalings to compensate for spherical divergence of the amplitudes.

In terms of image transforms, it is generally possible to choose any transform that leads to sparsity of the image in the transform domain, meaning the image can be well approximated by a small subset of coefficients (Figure 3.3). For seismic images, we are interested in local details, such as edges and singularities, which can be captured by wavelets or first differences. However, curvelets are not only multi-scale (like wavelets), but also multi-directional and thus are able to capture both point and line singularities, as well as smoothness along curved reflectors [22, 59]. As such, the curvelet transform is able to preserve structures in seismic images, even for a small number of coefficients, while sparse approximations of the original image coefficients lead to gaps in the subsurface structures (Figure 3.3). Another structure preserving transform that has been successfully used in the context of sparse seismic imaging, is the seislet transform [24, 60]. The seislet transform requires an estimation of the local slopes of events using plane wave deconstruction, while the curvelet transform detects dips automatically [22].

The objective function in equation 3.13 is a modified formulation of the basis pursuit denoise (BPDN) problem [54] and consists of a combined ℓ_1- and ℓ_2-norm, where λ is a trade-off (penalty) parameter that balances the two terms. The combined ℓ_1- and ℓ_2-norm is referred to as an elastic net in machine learning and has the effect of making the objective function strongly convex [61]. This allows us to optimize equation 3.13 with the linearized Bregman method, a simple to implement solver with few hyper parameters [62, 63], in which the penalty parameter λ plays a fundamentally different role than in comparable algorithms such as iterative soft thresholding (ISTA). A comparison of these two algorithms in the context of seismic imaging and the role of the thresholding parameter can be found in [41]. Practically, the ℓ_2-norm of the curvelet coefficients has no direct influence on the final image and in fact, for a large enough value of λ, the solution of equation 3.13 is equivalent to the solution of the BPDN problem, which is the same problem without the

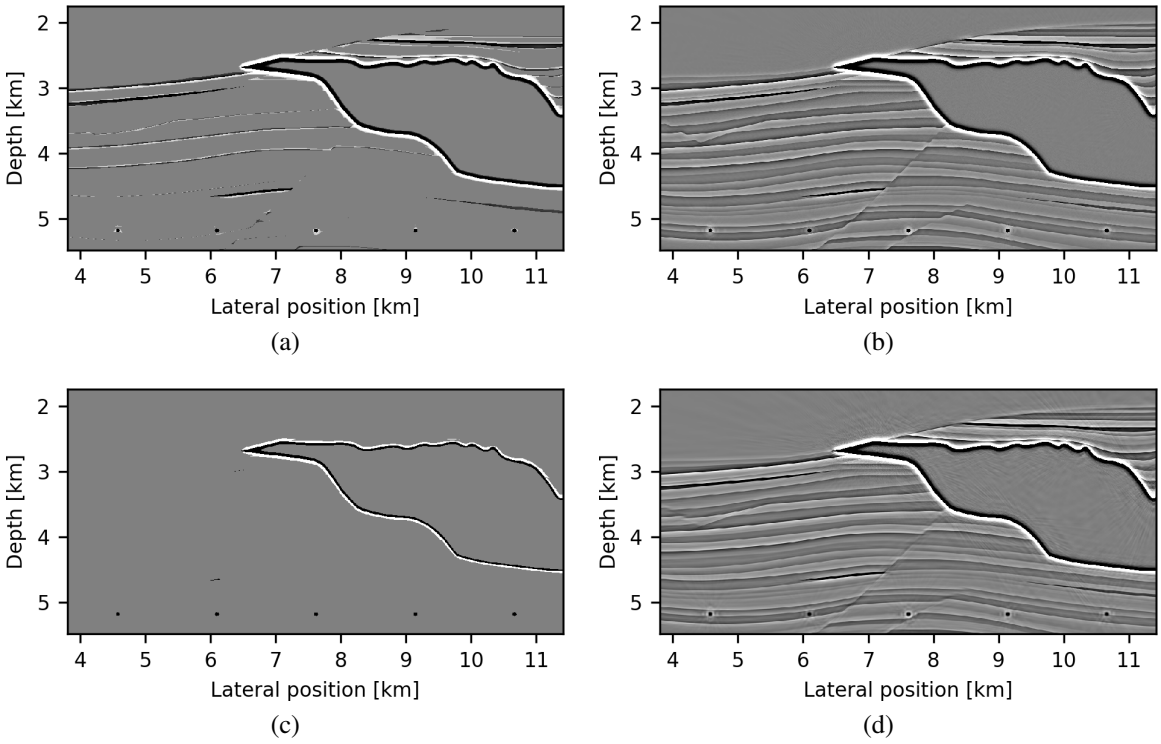


Figure 3.3: Sparse approximations of the true image in the image domain itself (left-hand column) and the curvlet domain (right-hand column). Figures (a) and (b) are sparse approximations using the largest 5 percent coefficients of the images in their respective domain and figures (c) and (d) are approximations using the largest 1 percent coefficients. The seismic image can be almost perfectly approximated by only 1 percent of the curvelet coefficients, as the sorted coefficients decay faster by magnitude than the original image coefficients. This makes the curvelet transform well suited for seismic imaging based on sparsity promotion.

ℓ_2 regularization term in the objective function. However, we have to include this term to make the objevtive function strongly convex and the ℓ_2-term has to act on the same coefficients as the ℓ_1-term. Strong convexity enables us to solve the problem with the linearized Bregman method, which has nicer numerical properties than the related ISTA algorithm. Furthermore, there exists theoretical justification for using the linearized Bregman method to solve overdetermined problems, such as LS-RTM, by working with random subsets of rows/measurements in each iteration [63]. In the extreme case of working with single rows (shots) of the linear operator and data, the method is then equivalent to the sparse Kaczmarz solver [64]. The linearized Bregman method is a fairly recent optimization algorithm, but

is closely related to older, well-established algorithms, such as the augmented Lagrangian method and the alternating direction method of multipliers [e.g. 62].

Sparsity-promoting LS-RTM in the frequency domain as a BPDN problem has been described, amongst others, in [19] and the adaption of the linearized Bregman method to this problem has been discussed in [41]. In contrast to the approach presented here, these publications solve Helmholtz equations rather than performing time-to-frequency conversion and are thus limited in their scalability to large-scale high frequency data sets. However, as the algorithm for solving equation 3.13 only differs in the way how the linearized data and gradients are computed, we refer to these publications for details on sparsity-promoting LS-RTM and the linearized Bregman method. For the sake of completeness and reproducibility, we include the algorithm for minimizing equation 3.13 with the linearized Bregman method in Algorithm 3.1. The method works for imaging velocity contrasts as well as acoustic impedance and basically consists of three simple steps: modeling the predicted linearized data (equation 3.3 or 3.12), computing the gradient by migrating the data residual (equation 3.7 or 3.10) and soft thresholding the curvelet coefficients of the updated image. Each iteration involves choosing a random subset of $\hat{n}_s \ll n_s$ sources and $\hat{n}_f \ll n_f$ frequencies for which the gradient is computed. The step length t can be chosen to be either constant or based on a dynamic update rule [61] (a constant step size was used in all numerical examples). The penalty parameter λ is set according to the maximum amplitude of the gradient in the first iteration, i.e.; $\lambda = c\|t_1\bar{\mathbf{g}}\|_\infty$ with $\|\mathbf{x}\|_\infty = \max(|x_1|, ..., |x_n|)$ being the infinity norm. The constant c determines how many coefficients pass the threshold in the first iteration. For $c = 1$, λ is set to the magnitude of the largest coefficient, causing no coefficient to pass the threshold in the first iteration. A smaller value of c, such as $c = 0.1$, results in a threshold that keeps all coefficients with magnitudes larger than $\frac{1}{10}$ of the maximum magnitude. In practice, this results in more coefficients entering into the solution early on. A detailed geophysical interpretation of each step of the algorithm is provided in Appendix A.3.

Algorithm 3.1 The linearized Bregman method for sparsity-promoting LS-RTM with randomized subsets of shots and frequencies. For each selected shot, a different subset of frequencies is selected; thus leading to a larger number of different frequencies in each image update. The algorithms consists of modeling the predicted linearized data $\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}}$ and migrating the data residual for obtaining the gradient. The image $\mathbf{x}_i$ is updated by applying the soft-thresholding function to the dual variable $\mathbf{z}_i$.

1. Initialize $\mathbf{x}_1 = \mathbf{0}$, $\mathbf{z}_1 = \mathbf{0}$, q, λ, batch sizes $\hat{n}_s \ll n_s$ and $\hat{n}_f \ll n_f$
2. **for** $i = 1, ..., n$
3. Select subset of shots and frequencies $\mathcal{S} = (\int_{\text{shot}}, \int_{\text{freq}}), |\int_{\text{shot}}| = \hat{n}_s, |\int_{\text{freq}}| = \hat{n}_f$
4. $\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}} = \mathbf{M}_l^{-1}\mathbf{J}_{\mathcal{S}}\mathbf{M}_r^{-1}\mathbf{x}$
5. $\bar{\mathbf{g}}_{\mathcal{S}} = \mathbf{M}_r^{-\top}\mathbf{J}_{\mathcal{S}}^{\top}\mathbf{M}_l^{-\top}\mathcal{P}_\sigma\big(\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}} - \bar{\mathbf{d}}^{\text{obs}}_{\mathcal{S}}\big)$
6. $\mathbf{z}_{i+1} = \mathbf{z}_i - t_i\bar{\mathbf{g}}_{\mathcal{S}}$
7. $\mathbf{x}_{i+1} = \mathbf{C}^{\top}S_\lambda(\mathbf{C}\mathbf{z}_{i+1})$
8. **end**

note: $S_\lambda(\mathbf{C}\mathbf{z}) = \text{sign}(\mathbf{C}\mathbf{z}) \cdot \max(0, |\mathbf{C}\mathbf{z}| - \lambda)$

$$\mathcal{P}_\sigma\big(\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}} - \bar{\mathbf{d}}^{\text{obs}}_{\mathcal{S}}\big) = \max\left(0, 1 - \frac{\sigma}{\|\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}} - \bar{\mathbf{d}}^{\text{obs}}_{\mathcal{S}}\|}\right) \cdot \big(\bar{\mathbf{d}}^{\text{pred}}_{\mathcal{S}} - \bar{\mathbf{d}}^{\text{obs}}_{\mathcal{S}}\big)$$

3.3 Numerical examples

In the following numerical examples, we will demonstrate that the method presented here, allows to perform least-squares migration at a fraction of the cost of conventional LS-RTM and without having to store or recompute time-domain wavefields. By using time-to-frequency conversion methods, the proposed algorithm does not rely on solving Helmholtz equations and scales to almost arbitrary model sizes and high frequencies. As part of our numerical examples, we will analyze the trade-off between memory usage and computational cost through varying the number of frequencies per iteration (frequency batch size) and compare it to time-domain imaging with optimal checkpointing. All of our numerical examples are computed with the Julia Devito Inversion (JUDI) framework [65], an open-source software package for seismic modeling and inversion based on Devito, a domain-specific language compiler for automated finite-difference computations [42, 43]. All wave equations were implemented using second order finite differences in time and 8th order finite differences in space, unless specificed otherwise. To simulate wave propagation in an infinite domain, we use simple absorbing boundary conditions (ABCs) using

a damping mask, as described in [66]. Optimal checkpointing in Devito is implemented through a Python wrapper [67] around the original Revolve library [27]. The results shown in this section are reproducible with JUDI and scripts are provided on Github [65]. The framework is implemented in the Julia programming language and uses a combination of distributed memory parallelism to parallelize over the shot locations and shared memory parallelism with OpenMP for the wave equation solves.

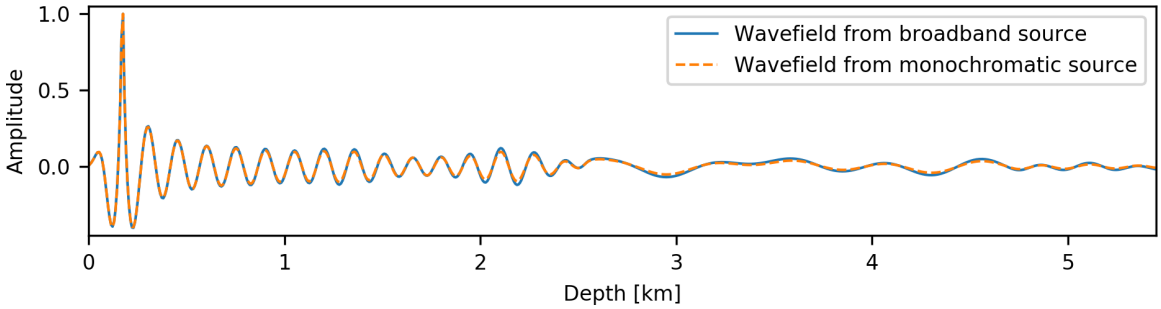
3.3.1 Sigsbee 2A

For our first numerical example, we use the Sigsbee 2A velocity model [68], a challenging salt model of 9.2 by 24.6 km. Due to the size of the model and the number of time steps that are required to propagate the wavefield to all parts of the domain, storing the forward wavefields in random access memory (RAM) can already be problematic. For our experiments, we model wave propagation for 10 seconds, which corresponds to $14,095$ time steps, using the time interval provided by the Courant-Friedrichs-Lewy (CFL) condition [69]. Storing $14,095$ wavefields in RAM as single precision arrays with a grid spacing of 7.62 m requires 237 GB of memory, while saving the wavefields in memory at a sampling interval of 4 ms ($2,500$ wavefields) requires 42 GB. These numbers can be prohibitively expensive, especially if we want to compute multiple gradients in parallel on the same computational node. For this reason, wavefields, or a compressed version of them, are typically written to secondary storage devices. Alternatively, optimal checkpointing and on-the-fly Fourier transforms allow us to compute the gradients using substantially less or memory or to write only a small subset of wavefields to disk. For a fair comparison, we fix the allowed amount of memory for both methods to 700 MB, which corresponds to saving 40 real-valued or 20 complex wavefields in RAM.

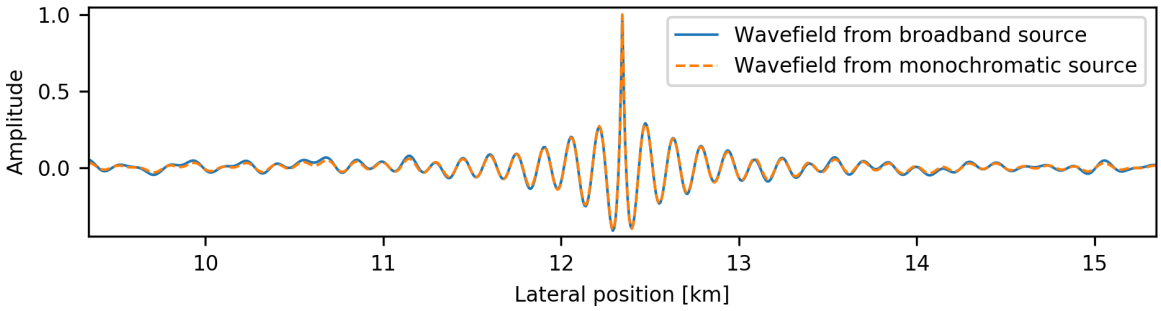
For practical purposes, we compute gradients with the full time-domain source wavelet and data residuals as forward and adjoint sources, rather than modeling with monochromatic sources/residuals as indicated by equations 3.3 and 3.6. In other words, instead of

performing Fourier transforms of the time-domain data, extracting the required frequencies and an inverse Fourier transform back to the time-domain, we use the unmodified time-domain wavelets and data residuals for modeling. This avoids having to extract monochromatic frequencies of the source wavelet and shot records, which is cumbersome if the frequencies do not lie on the corresponding time axis and therefore need to be interpolated. Overall we only need to perform two on-the-fly DFTs per frequency to compute the gradient for one shot record: one for the forward wavefield and one for the adjoint wavefield. Using the broadband wavelets and shot records as sources is possible, as extracting a monochromatic Fourier-domain wavefield for a given frequency from its corresponding monochromatic source, yields the same result (up to a constant) as modeling with the full time-domain source (Figure 3.4). Strictly speaking, this modification destroys the exact adjoint property of our demigration-migration operator pair, since we inject additional energy, but the introduced error is purely a scaling error and does not affect the position of reflectors.

In our first numerical experiment, we compare time-domain SPLS-RTM with optimal checkpointing and frequency-domain SPLS-RTM with on-the-fly Fourier transforms. The data set consists of 935 observed shot records with 10 seconds recording time and a peak frequency of 15 Hz and maximum frequency of 40 Hz. We simulate a marine streamer acquisition with 100 m minimum offset, 12 km maximum offset and $1,200$ evenly spaced hydrophones. We perform 20 iterations of the linearized Bregman method with 100 randomly selected shots per iteration (with replacement), which corresponds to approximately two passes trough the data. This means, in expectation, every shot record is migrated only twice. We found that the effect of the trade-off between batch size and number of iterations is negligible, as long as we avoid either extremes, i.e. using a very small batch size or very few iterations. For frequency-domain SPLS-RTM with on-the-fly Fourier transforms, we randomly select 20 different frequencies for each shot and each iteration. The frequencies are selected from a continuous frequency band between 3 and 40 Hz, according to the spec-



(a)



(b)

Figure 3.4: Comparison of monochromatic wavefields computed with on-the-fly DFTs using the full broadband source wavelet or the corresponding monochromatic source. The plots show vertical (a) and horizontal (b) slices of a monochromatic 10 Hz wavefield from the Sigsbee 2A model.

trum of the source wavelet. For this, we convert the frequency spectrum of the source to a cumulative probability function, generate uniform random values between 0 and 1, and select the corresponding frequencies on the $x-$axis of the probability function. This strategy ensures that a large number of random frequencies approximates, in expectation, the full spectrum of the source wavelet. Alternatively, the spectrum of the data can be used for this process, if the source wavelet has not been estimated prior to migration. The noise level σ in the algorithm was set to zero, since the observed data is noise free and a constant step size t was used for all SPLS-RTM examples. For every run, we used the largest possible step size that preserves numerical stability of the modeling scheme during all iterations. As a reference for our results, we also compute the time- and frequency-domain RTM images, which correspond to one full data pass, since every shot record is migrated once. The RTM and SPLS-RTM results for the frequency domain are shown in Figure 3.5 and a

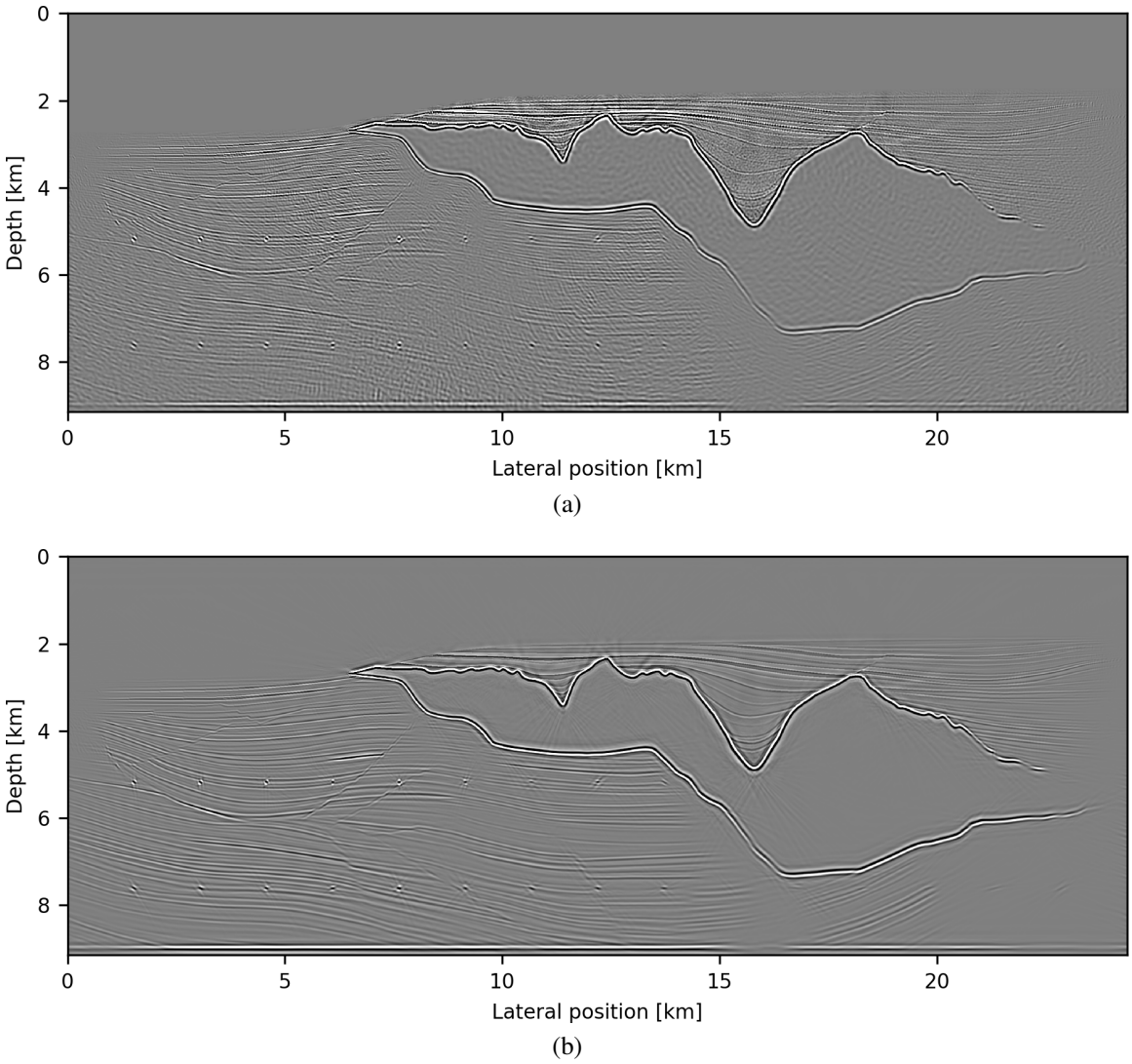


Figure 3.5: Reverse-time migration with 20 randomly selected frequencies per shot (a), in comparison to sparsity-promoting LS-RTM after 20 iterations, using 100 shots with 20 frequencies per iteration (b). With only two passes through the full dataset, SPLS-RTM is able to remove the noise from frequency randomization, as well as the imprint of the source wavelet. The only post-processing that was applied to the results, is a linear depth scaling.

close-up comparison of all images is provided in Figure 3.6. While the frequency-domain RTM image is noisy due to frequency-subsampling artifacts, SPLS-RTM is able to map the incoherent noise to a coherent image and provides the same high-quality image as the time-domain method. The only post-processing that was applied to the results, is a linear depth scaling, to emphasize deeper reflectors.

As mentioned, the amount of memory for both optimal checkpointing and on-the-fly

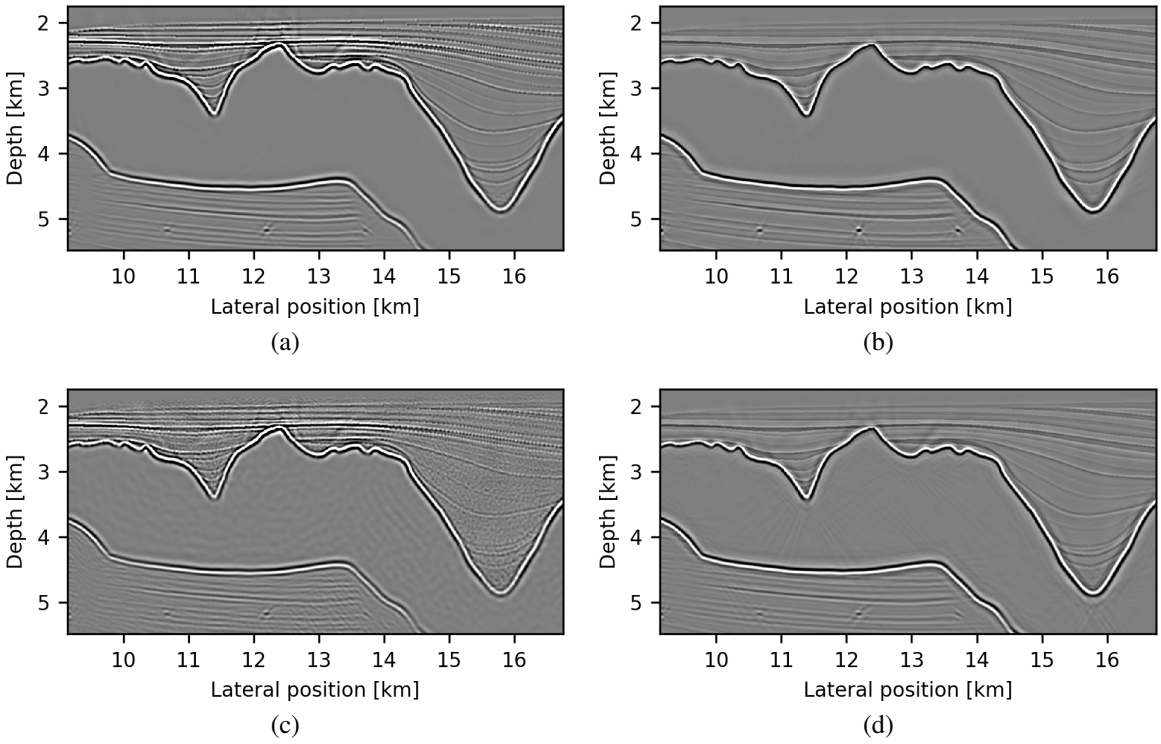


Figure 3.6: A close-up comparison of the images from time-domain RTM (a), time-domain SPLS-RTM (b), frequency-domain RTM (c) and frequency-domain SPLS-RTM (d). While RTM with on-the-fly Fourier transforms and randomized subset of frequencies leads to a noisy image, sparsity-promoting LS-RTM is able to convert noise to coherent reflectors and provide the same high-fidelity image as time-domain SPLS-RTM with optimal checkpointing. However, due to the limited number of iterations, not all energy is converted back into coherent energy, as apparent by the slightly weaker diffractors in (d). A comparison between the additional computational cost of checkpointing and on-the-fly DFTs is provided in the discussion.

Fourier transforms is fixed to 700 MB in the previos experiments (40 real-valued or 20 complex wavefields). For optimal checkpointing, the amount of memory defines the trade-off between the number of checkpoints and the computational cost for recomputing wavefields. Decreasing the memory increases the computational cost, as fewer checkpoints can be saved and more wavefields need to be recomputed. For imaging with on-the-fly Fourier transforms however, this relationship does not apply. Decreasing the available memory decreases the number of wavefields that can be stored, but it also decreases the amount of computations, since less on-the-fly DFTs have to be computed. However, in this case, the

trade-off is related to the amount of frequency subsampling artifacts in the images and to how many iterations of SPLS-RTM have to be performed to achieve the same quality of the final image. To demonstrate this relationship, we carry out a second numerical experiment in which we perform frequency-domain SPLS-RTM with on-the-fly DFTs using only 10 randomly selected frequencies per shot instead of 20. As expected, the convergence of the SPLS-RTM data misfit for 10 frequencies is considerably slower than for 20 frequencies and more iterations are necessary to bring the misfit of the current subset of shots and the image error down to a comparable level (Figure 3.7). On the other hand, each iteration requires only half the amount of memory and half the number of DFTs, which decreases the runtime for computing gradients. This is illustrated in Figure 3.8, in which we plot the time-to-solution for computing the gradient for a single shot record with on-the-fly DFTs as a function of the number of frequencies. For comparison, we also provide the runtime for computing a gradient using optimal checkpointing. All timings were obtained using a single CPU with 10 threads. A detailled description of the configuration and utilized hardware is given in Appendix A.4.1.

Furthermore, we demonstrate the effect of varying the batch size of shots versus the batch size of frequencies. For this, we repeat the previous experiment, but using twice as many shots, while keeping the numbers of frequencies fixed to 10. As evident from the convergence plots of the data residual and model error (Figure 3.7), increasing the batch size to 200 with a frequency batch size of 10 yields almost identical results as the example with 100 shots and 20 frequencies. This once again emphasizes, that the quality of the final results is similar if the product of number of iterations, shots and frequencies is kept constant. This observation is important, as it provides the possibility to adapt the optimization parameters (batch size, number of data passes) to the available computational resources. For example, if many computational nodes are available, we can increase the batch size of shots and decrease the number of frequencies, whereas we can decrease the batch size and increase the number of frequencies if only few nodes are available. Over-

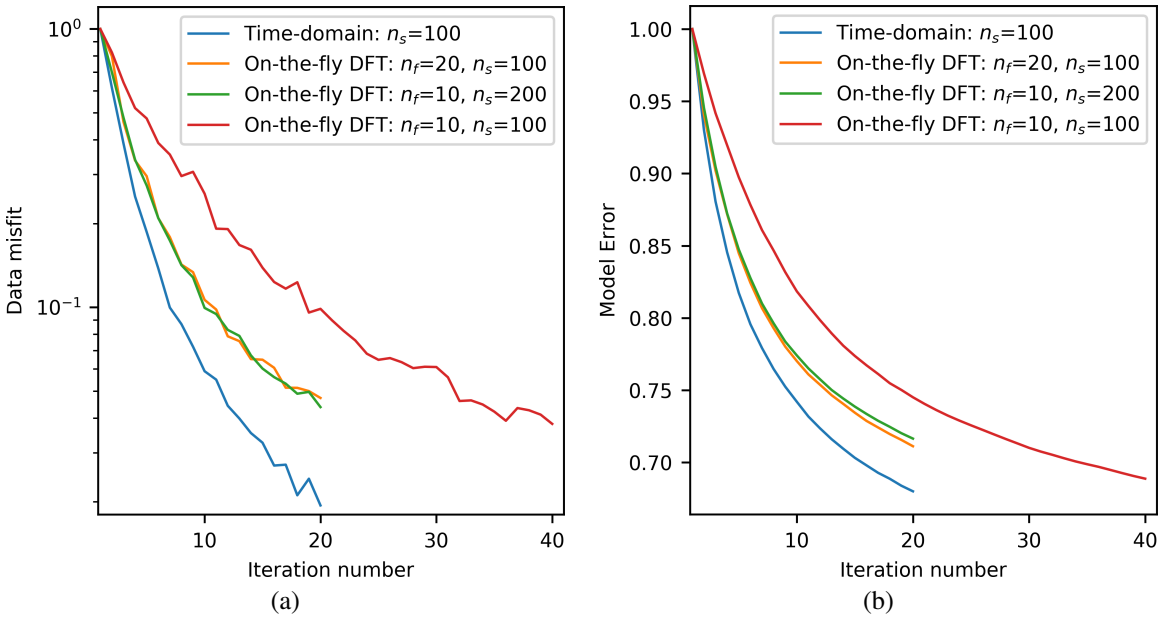


Figure 3.7: Normalized ℓ_2-norm data misfit (a) and ℓ_2-norm reconstruction error (b) for compressive LS-RTM in the time domain and with on-the-fly DFTs, using the linearized Bregman method. For a smaller number of frequencies n_f, more iterations have to be performed to reduce the data misfit to a comparable level, but each iteration requires less memory and computations. Keeping the product of the batch size of shots n_s and frequencies n_f constant, yields results of comparable quality.

all the time-domain result has the smallest data residual and model error, but also comes at higher computational cost, since all all wavefields have to be saved or reconstructed for computing the gradient. The convergence rate of the linearized Bregman method has not been analyzed in the literature, but is linear at best. Standard stochastic optimization methods, such as stochastic gradient descent, have a sub-linear convergence rate $\mathcal{O}(1/n)$, with n being the iteration number. However, in practice, the sublinear convergence rate is mostly problematic at very high iteration numbers when $\frac{1}{n}$ is very small and we want to solve the problem to convergence. During early iterations, the behavior of the algorithm is mostly dominated by a constant (algorithm-dependent) factor $\mathcal{O}(1)$, rather than the asymptotic behaviour, making the algorithm effective when only a small number of data passes is affordable.

In our final experiment using the Sigsbee 2A model, we investigate the sensitivity of the

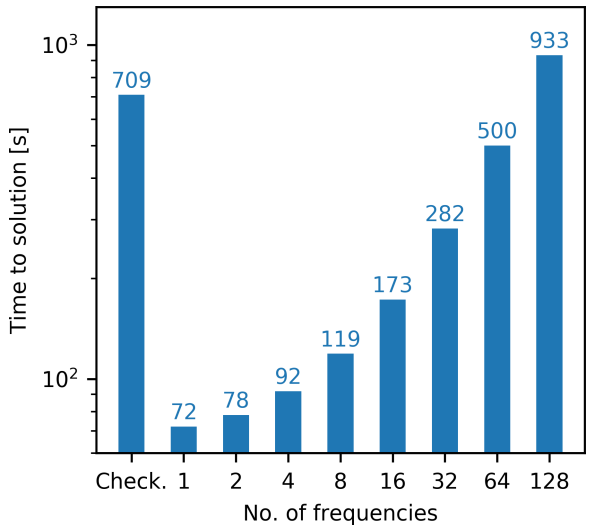


Figure 3.8: Timings for computing the gradient of the full Sigsbee model for a single shot record as a function of the number of frequencies and in comparison to optimal checkpointing (leftmost bar).

results to the choice of the regularization parameter λ (i.e. the soft thresholding value) and the effect of sparsity-promoting techniques to weak events in the seismic image. For this, we repeat the previous SPLS-RTM experiments with on-the-fly DFTs and a fixed number of shots and frequencies, but with varying values of λ. Namely, we use 100 randomly selected shots and 20 randomly selected frequencies per iteration with a total number of 20 iterations. In the previous examples, we chose the thresholding parameter through parameter testing, such that after the maximum number of iterations, most reflection events were recovered, but no incoherent noise was present in the final image. We now conduct two additional experiments in which we set $\lambda = 0$ (no sparsity promotion) and $\lambda = 4e-4$ (stronger sparsity promotion than before), while previously, we used a value of $\lambda = 1e-4$.

The effect of sparsity promotion and the choice of the thresholding parameter on two areas of the Sigsbee model is shown in Figures 3.9 to 3.11. Figure 3.9 shows a close-up view of the top-of-salt region, in which we have the best illumination in the model. Figure 3.9b shows the result using no sparsity promotion, which means that no thresholding is applied to the coefficients. Even though no sparsity-promotion was used, the result shows a very high signal-to-noise ratio, since most of the incoherent noise stacked out over the course

of the inversion. Furthermore, we can observe that in the cases where sparsity promotion was used, the final results are expectedly not very sensitive to the choice of λ (Figures 3.9c and 3.9d). However, the situation is different for areas with poor illumination, such as in the sub-salt region shown in Figure 3.10. Here, we can observe that no sparsity promotion leads to a worse signal-to-noise ratio (Figure 3.10b), but that the result is also more sensitive to λ. Namely, a larger value of λ (i.e. stronger thresholding) removes weak events such as the diffractors (Figure 3.10c and 3.10d). This observation is further emphasized in one-dimensional well-log comparisons (Figure 3.11), which were extracted at 12.1 km horizontal position.

It is important to consider, that we use a fixed number of data passes and therefore iterations for our examples, and that running a sufficiently large number of iterations will eventually recover the missing coefficients. However, for a larger value of λ, more iterations are necessary to recover the small coefficients, while choosing λ too small, causes incoherent noise to enter the solution after a few iterations. The right choice of λ is therefore a trade-off between noise, missing coefficients and the number of iterations. Using a smaller batch size of shots, does not inherently increase the incoherent noise, but leads to a poorer illumination and therefore increases the sensitivity to the choice of λ and the likelihood that coherent signal is accidentally removed or that noise in introduced into the solution.

3.3.2 BP Synthetic 2004

The results for the Sigsbee 2A model were obtained under ideal conditions, in which the noise-free observed data was generated with the same linearized modeling operator that was used for the inversion (inverse crime). To demonstrate that our proposed approach also works in a more realistic setting and scales to large-scale models, we test our algorithm on the BP synthetic 2004 model [70]. The model has a size of 11.9 by 67.4 km and is interpolated to a grid spacing of 6.25 m ($1,911 \times 10,789$ grid points). We generate the

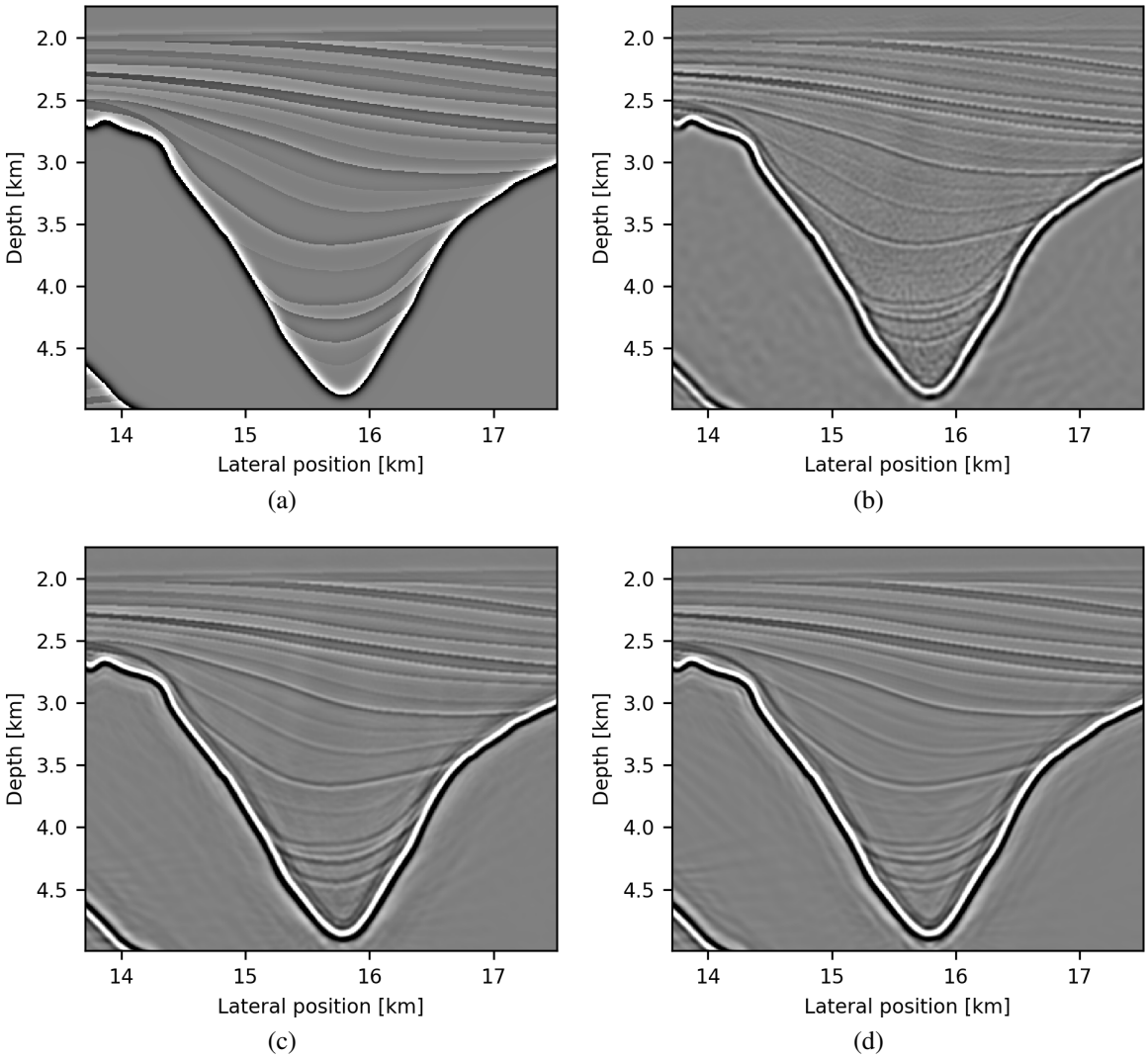


Figure 3.9: Close-up views of the top-of-salt region for different values of the regularization parameter λ. Figure (a) is the true image, (b) is the result for $\lambda = 0$ (no sparsity-promotion), (c) is the result for $\lambda = 1e-4$ and (d) for $\lambda = 4e-4$. Since the top-of-salt region is well illuminated, the resulting image is not very sensitive to the choice of the thresholding parameter and the effect of sparsity promotion is less apparent than in the sub-salt area.

observed data with the same acquisition geometry as the original data released by BP, using a 15 km streamer, 12 seconds recording time and $1,340$ shot locations. However, unlike the original data, we model the data without surface-related multiples and with a peak frequency of 20 Hz instead of 27 Hz. To provide a non-inversion crime setting, we model the data with the acoustic forward modeling operator and not with the demigration operator. We generate the data using the true velocity and density models, whereas for

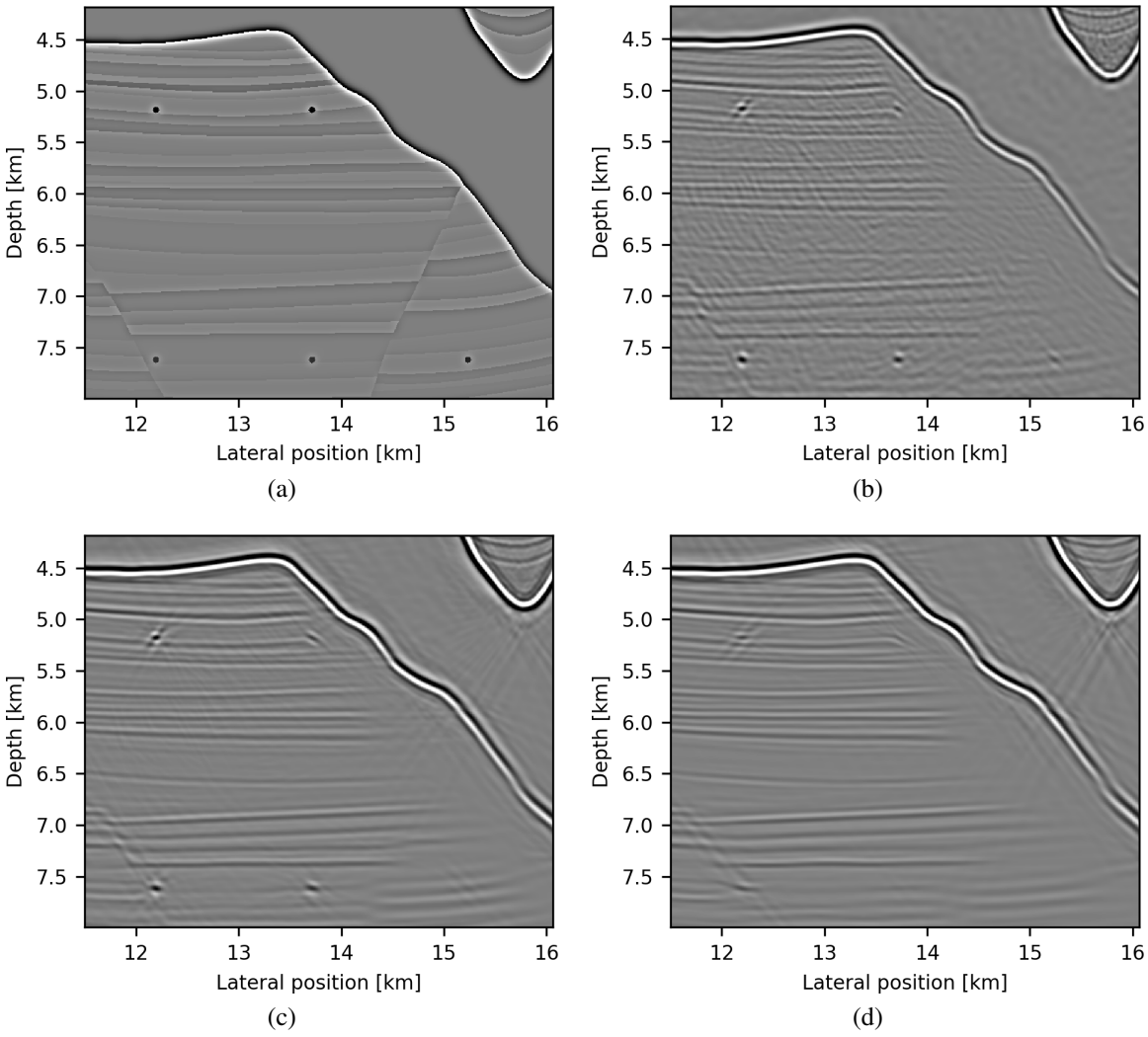


Figure 3.10: Close-up views of the sub-salt region of the true image (a) and results after 20 iterations of SPLS-RTM with the linearized Bregman method using $\lambda = 0$ (b), $\lambda = 1e-4$ (c) and $\lambda = 4e-4$ (d). Compared to the top-of-salt area, the benefit of sparisty-promotion is greater, but the result is also more sensitive to the choice of λ.

inversion, we only use a smooth migration velocity model, but no density. Furthermore, we model the observed data with a 16^{th} order finite-difference (FD) stencil, while using an 8^{th} order FD stencil for the inversion. Changing the discretization order is easily possible in JUDI, as the software uses Devito to automatically optimize and generate completely new source code for solving a specified wave equation in every individual run [71, 72]. Since we use a different acoustic wave equation without density for inversion and change the finite-difference stencil order, this leads to different sets of source code for modeling

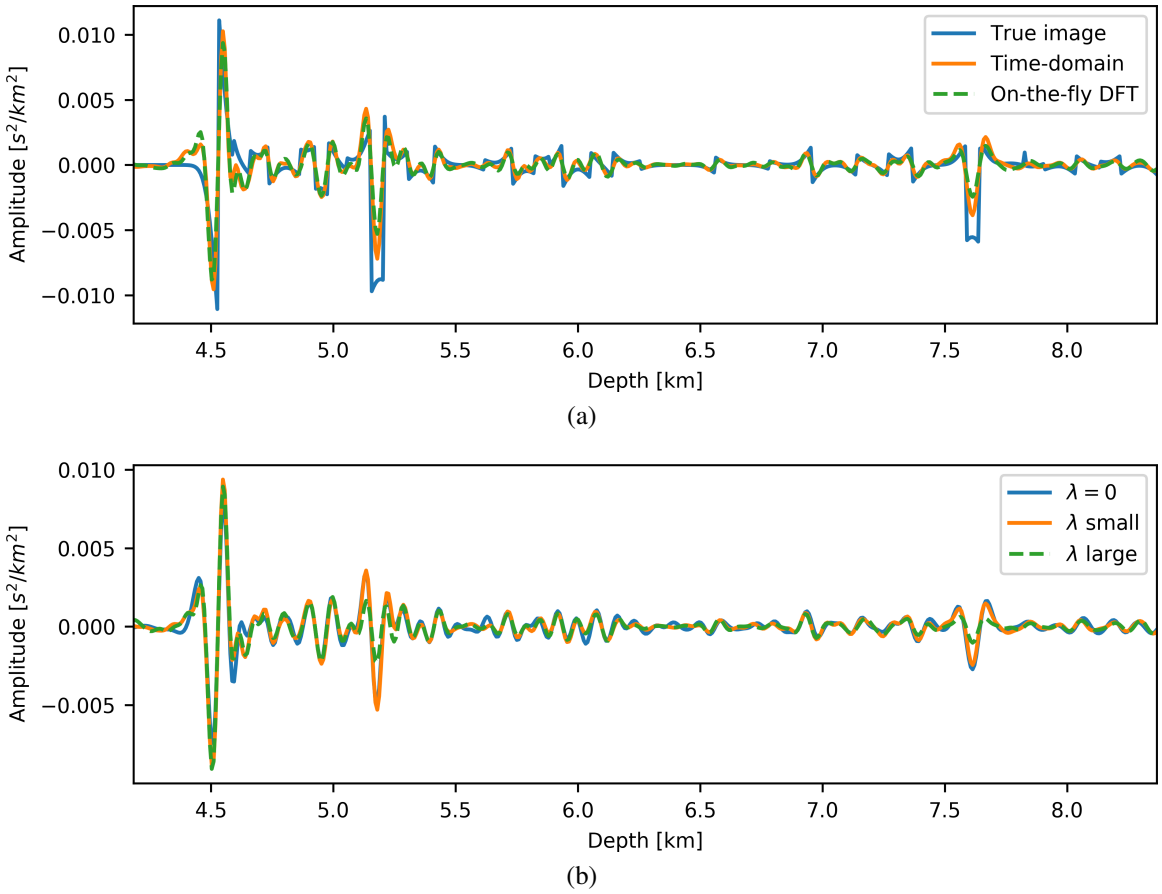


Figure 3.11: Trace comparisons of the results after 20 iterations of SPLS-RTM using time-domain modeling and on-the-fly DFTs (a) and for different values of λ (b).

the observed data and running SPLS-RTM. Furthermore, we add Gaussian noise to all observed shot gathers with a signal-to-noise ratio of 17 dB (8.13 dB after muting the direct wave).

As before, we compare frequency-domain RTM with on-the-fly DFTs and 20 randomly selected frequencies per shot to frequency-domain SPLS-RTM with randomized shots (Figures 3.12 and 3.13). For this example, we run the inversion for 20 iterations, using 200 randomly selected shots per iteration with 20 frequencies per shot. As for the Sigsbee example, the frequencies are selected from a continuous frequency band between 3 and 45 Hz, using the spectrum of the source wavelet as the probability distribution. As a reference solution, we also compute the time-domain SPLS-RTM image using optimal checkpointing (Figure 3.13 a). As indicated by a close-up comparison of the results, SPLS-RTM is

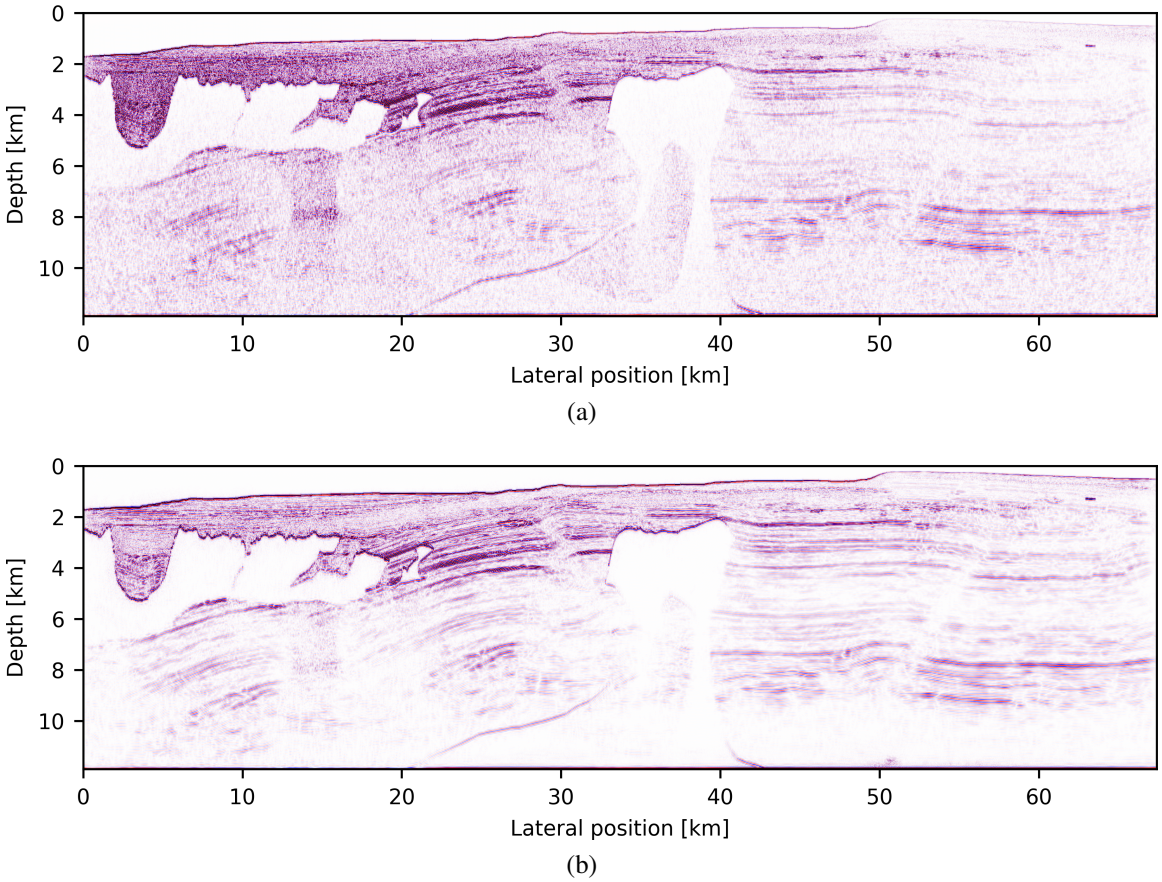


Figure 3.12: Comparisons of frequency-domain RTM (a) and SPLS-RTM (b) using on-the-fly Fourier transforms with 20 randomly selected frequencies per shot record. The SPLS-RTM image is shown after 20 iterations of the linearized Bregman method, using 200 random shots per iteration, which corresponds to three passes through the data.

once gain able to map the frequency subsampling artifacts in the RTM image into coherent energy and provide an image of similar quality as the time-domain method (Figures 3.12 – 3.15). Since the observed data is not generated with the demigration operator that is used for inversion and is generated using a density model, the amplitudes of the predicted data can never exactly match the amplitudes of the observed data, which is why the data misfit for the current subset of shots only decays by 32 percent (Figure 3.16a). This amplitude mismatch in combination with a smooth migration velocity model without sharp salt boundaries is also responsible for the the slight blur of the salt dome's top reflector.

As in our Sigsbee example, we measure the time-to-solution for computing the gradient for a single shot record as a function of the number of frequencies (Figure 3.16b).

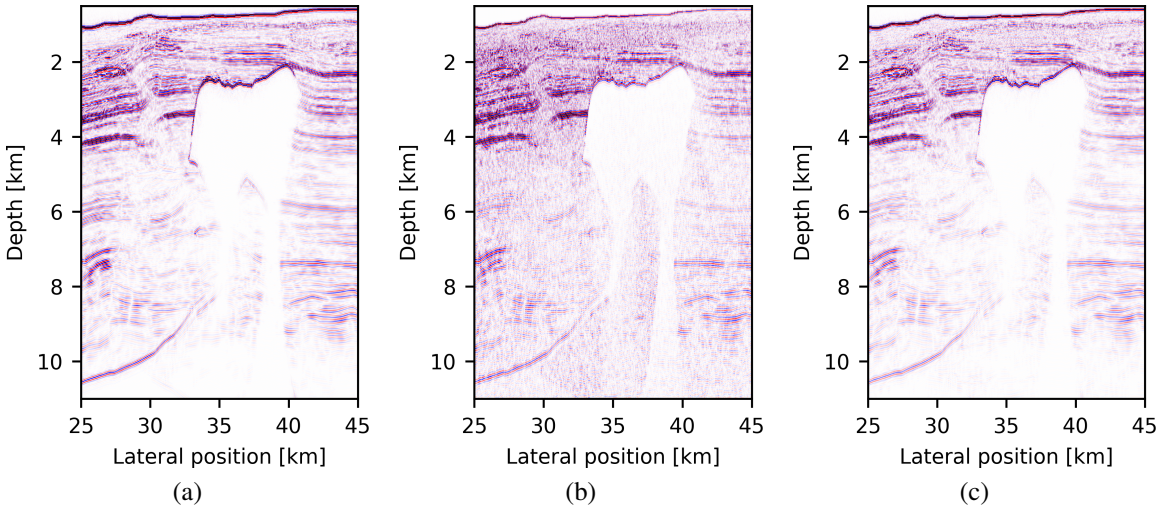


Figure 3.13: Close-up comparison of time-domain SPLS-RTM with optimal checkpointing (a), frequency-domain RTM (b) and frequency-domain SPLS-RTM (c). The results for frequency-domain RTM and SPLS-RTM were computed with 20 randomly selected frequencies per shot record. A depth scaling was applied to the RTM image, to make up for the lack of the depth-scaling pre-conditioner that was used for SPLS-RTM.

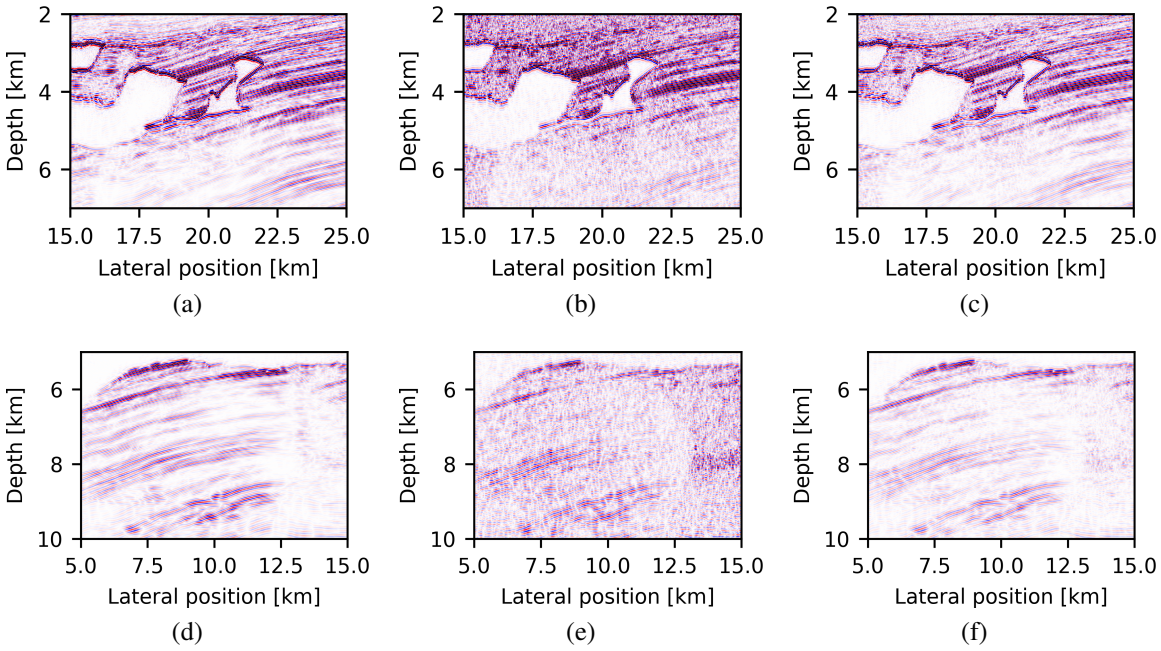


Figure 3.14: Close-up views of the time-domain SPLS-RTM result (a, d), frequency-domain RTM (b, e) and frequency-domain SPLS-RTM (c, f). The top row shows a shallow part of the image with good illumination in comparison to the sub-salt area, which is affected stronger by frequency subsampling (bottom row).

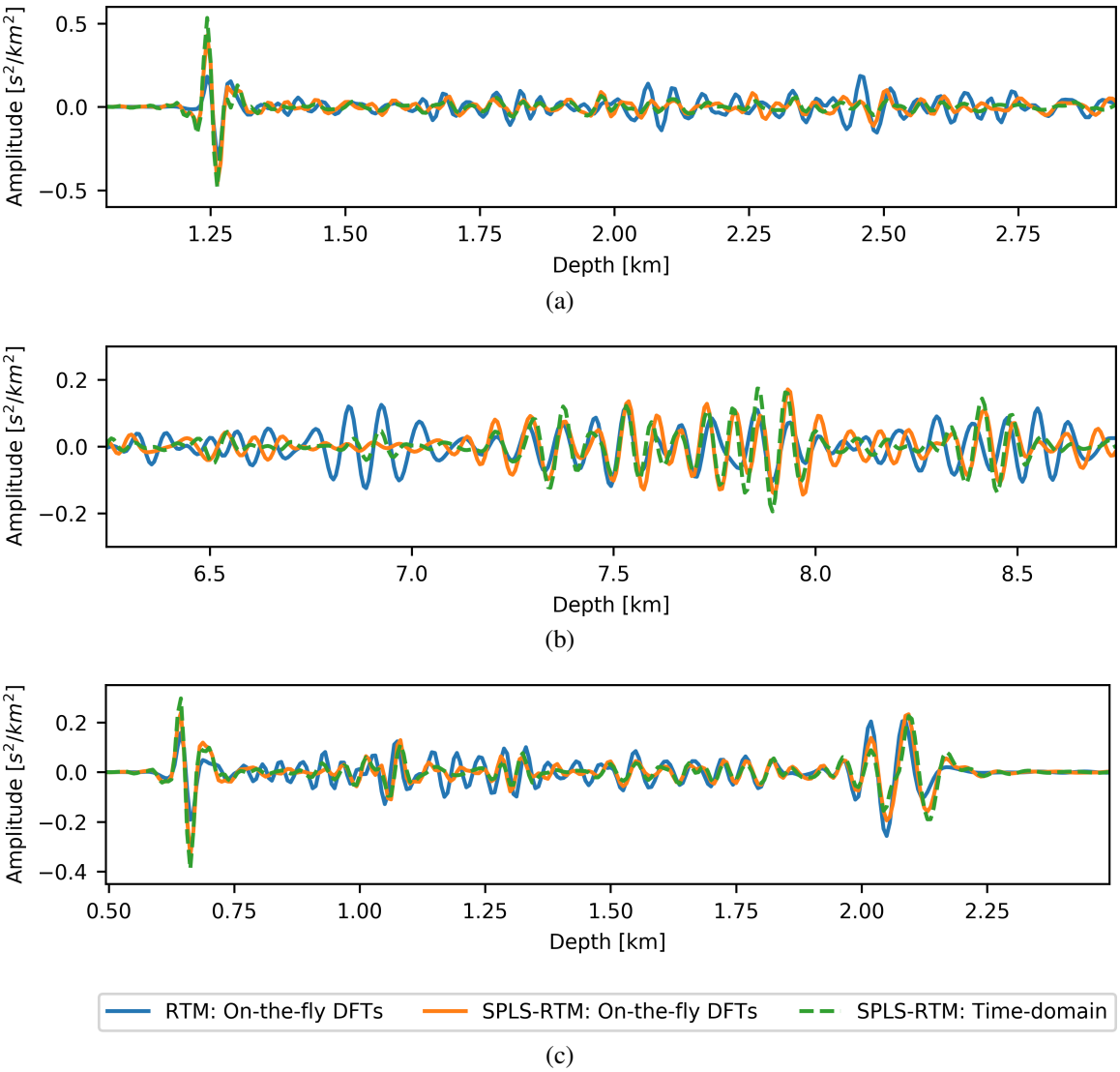


Figure 3.15: Trace comparisons of the imaging results at various locations of the model. Plots (a) and (b) are well-log plots of different depths at 10 km lateral position. The traces in (c) were extracted at 40 km lateral position.

The timings are obtained with the same computational set up as before, using a single Intel Xeon CPU with 10 cores (Appendix A.4.1). In this particular case, computing one gradient with optimal checkpointing takes longer than computing a gradient for 64 frequencies with on-the-fly DFTs, but less time than 128 frequencies. In the RTM and SPLS-RTM example, we only use 20 randomly selected frequencies per shot record, making the computation of a single gradient approximately three times faster than the computation of a time-domain gradient with optimal checkpointing and the same amount of computational

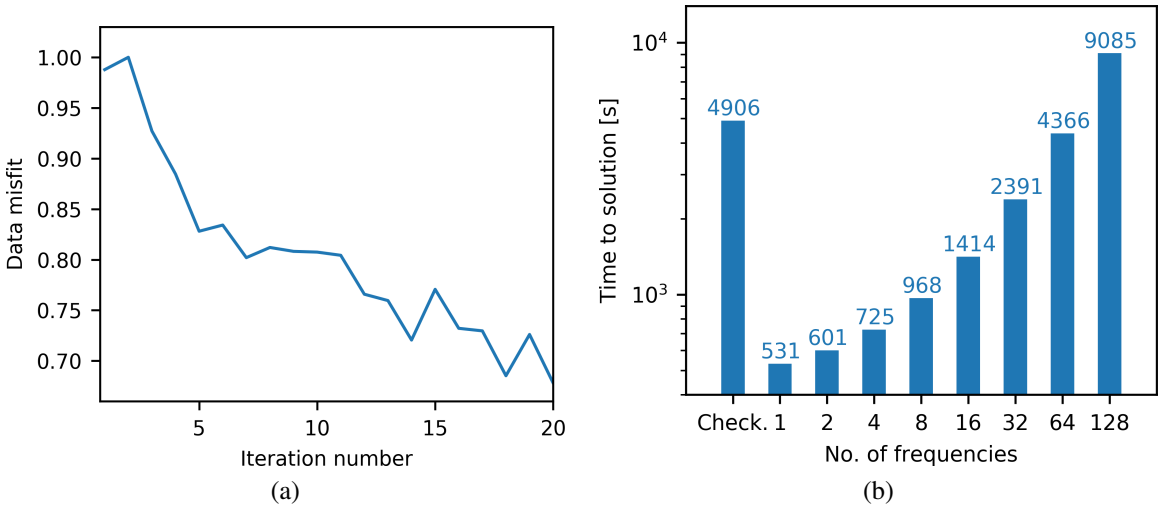


Figure 3.16: Relative data misfit for the current subset of shots during SPLS-RTM with on-the-fly DFTs (a) and timings for computing the gradient of the BP model for one shot record as a function of the number of frequencies (b). The bar on the left-hand side denotes the corresponding time-to-solution using optimal checkpointing.

resources. While the frequency-domain SPLS-RTM result using only 20 randomly selected frequencies per shot is considerably faster than the its time-domain equivalent, it also still exhibits a slight amount of low-amplitude noise, which underlines the trade-off between number of frequencies and quality of the result that is inherent to this approach. While increasing the number of frequencies yields a result that is closer to the time-domain image, it also diminishes the computational speed up in comparison to optimal checkpointing.

3.4 Discussion

The main challenges of least-squares reverse-time migration are the large number of shots that have to be migrated in each iteration of gradient-based optimization algorithms, as well as the necessary access to the forward wavefields in reverse order for computing the gradient. A straight-forward implementation of time-domain LS-RTM is to forward propagate the source wavefield for all time steps, store the wavefields in memory and access them in reverse order during the adjoint time loop. The obvious drawback of this approach is that the required memory grows linearly with the number of time steps and the approach quickly

Strategy	Memory	Additional cost
TD: save all wavefields	$\mathcal{O}(n_t)$	-
TD: optimal checkpointing	$\mathcal{O}(\log n_t)$	$\mathcal{O}(\log n_t)$
TD: boundary reconstruction	$\mathcal{O}(n_t)$	$\mathcal{O}(n_t)$
FD: on-the-fly DFT	$\mathcal{O}(n_f)$	$\mathcal{O}(n_f)$

Table 3.1: Asymptotic behaviour of memory requirements and additional computational cost for different strategies to compute adjoint-state gradients in the time domain (TD) and frequency domain (FD). The total number of model grid points in this analysis is assumed to be constant and is therefore excluded from the analysis. However, while reconstructing wavefields from the boundary scales linearly with the number of time steps, it requires substantially less memory than saving the full wavefield (i.e. the asymptotic behavior has a smaller constant). The analysis in this table holds for both 2D and 3D domains.

becomes infeasible for any realistically sized models and recording times. Similarly, saving and reconstructing the wavefields from the boundary scales linearly with the number of time steps as well, but the asymptotic behavior has a smaller constant than saving the full wavefields, as the wavefield is only saved in a subset of the domain. Alternatively, the problem can be addressed by storing only a small subset of forward wavefields and by recomputing the in-between wavefields from the last checkpoint during the reverse time loop. Checkpointing therefore provides the user with a possible trade-off between memory usage and the number of time steps that have to be recomputed, which is captured in the recomputation ratio. This parameter is given by the total number of time steps (including recomputations) divided by the original number of forward time steps. In an important series of publications, [27] describe an algorithm for computing the *optimal* trade-off between these quantities and show that the amount of required memory and the recomputation ratio for optimal checkpointing grow logarithmically with the number of time steps n_t (Table 3.1).

For compressive imaging with on-the-fly Fourier transforms, the asymptotic behavior of the required memory and additional computations is fundamentally different, as these quantities grow as a function of the number of frequencies and not with the number of time steps (Table 3.1). Both memory and computational cost grow linearly with the number of

frequencies and therefore have worse asymptotic behavior than optimal checkpointing, but they are independent of the number of time steps. This leads us to the critical question of how many frequencies n_f are required for computing the gradient. One could argue that for a fair comparison, the number of frequencies should be equal to the number of computational time steps—i.e. the LS-RTM gradient should be computed as the sum over all $n_f = n_t$ frequencies, as this yields the same gradient as in the time-domain, where the gradient is computed as a sum over all computational time steps. However, our numerical experiments have shown that, in practice, we can get away with a much smaller number of frequencies than time steps and still achieve satisfactory imaging results, if we cast LS-RTM as an ℓ_1-norm minimization problem. Furthermore, our examples show that the number of frequencies influence the convergence of the algorithm and determine how many iterations we need to run to obtain results of comparable quality. In general, computing the gradient with a smaller subset of frequencies leads to stronger subsampling artifacts and requires a more aggressive thresholding of the image's curvelet coefficients to remove the noise, which in turn increases the necessary number of iterations. However, a quantitative relationship between the number of frequencies, the amount of noise and the required number of iterations is at this point not available and will require further investigations. A possible reference point is the compressive sensing literature itself, which provides theoretical relationships between the sparsity of a signal, the amount of necessary measurements and the reconstruction error. In particular, [54] shows that the the number of required measurements grows with $\mathcal{O}(n_s \log n_l)$, where n_s is the number of most important coefficients of the signal in some transform domain and n_l is the signal length. For compressive imaging with on-the-fly DFTs, this means that the number of necessary frequencies and shots will depend on the sparsity of the unknown image and the number of gridpoints, but not on the number of time steps.

In our analysis (Table 3.1), we assume that the number of grid points is fixed, but it is worth mentioning that the methods have different asymptotic behaviors as a function of

the model size. Namely, assuming we have n grid points in each dimension, the boundary reconstruction method scales with $\mathcal{O}(n)$ in 2D and $\mathcal{O}(n^2)$ in 3D, while all other methods scale with $\mathcal{O}(n^2)$ in 2D and $\mathcal{O}(n^3)$ in 3D. In practice, the asymptotic behavior of the memory requirements can only serve as a guideline, as it describes its limiting behavior and the question of which approach requires the least amount of memory, needs to be evaluated on a case by case basis and depends on the specific number of dimensions, time steps and grid points. However, for a large enough number of time steps, optimal checkpointing will eventually always require less memory than the saving the wavefields at the boundary.

The second important question is how much additional computations have to be carried out for a given amount of memory, which will determine how fast the two approaches perform in practice. In our first numerical example, we fix the available memory for both on-the-fly DFTs and optimal checkpointing to 700 MB, which corresponds to 20 frequency-domain wavefields or 40 time-domain checkpoints. The recomputation ratio with 40 checkpoints for $14,095$ time steps is estimated by the Revolve library as 3.06, which means that approximately $2n_t$ additional forward time steps have to be modeled for computing a gradient [27]. For time-to-frequency conversion with on-the-fly DFTs, the main additional computational cost results from multiplying the time-domain wavefield of the current time step with the sine and cosine terms of equation 3.7. Multiple frequencies can be computed in a single time loop, but result in an inner loop over the number of frequencies at each time step. The additional computational cost for on-the-fly DFTs therefore depends on the number of frequencies and how the DFT is implemented. Evaluating trigonometric functions is generally expensive, but since the sine and cosine terms in the equation 3.7 are not spatially varying, they can be precomputed and used for all grid points. Furthermore, it is possible to compute the DFT on a coarser time grid than the computational time axis (e.g. at the Nyquist rate), which reduces the number of floating point operations. For optimal checkpointing, the computational cost of remodeling a given number time steps depends on the type of wave equation and the order of the finite-difference (FD) stencil.

Modeling anisotropic or elastic wave equations is more expensive than solving acoustic wave equations and the amount of floating point operations increases further for higher order FD stencils. On the other hand, on-the-fly DFTs are independent of the discretization order (at least for a fixed grid spacing), but they also become more expensive for wave equations with coupled wavefields, as the cost increases linearly with the number of DFTs that have to be computed for each wavefield. In our numerical examples with the Sigsbee 2A and BP Synthetic 2004 model, we found that computing the gradient for one source using optimal checkpointing with $\log n_t$ checkpoints, was timewise equivalent to computing a frequency-domain gradient with approximately 70 to 100 frequencies. Since we only used 10 or 20 frequencies per shot record for our imaging examples with on-the-fly DFTs, we were able to reduce the time-to-solution per gradient by a factor of 3 to 4 in comparison to optimal checkpointing, with the same amount of computational resources.

Regarding the effects of sparsity-promoting minimization in the context of seismic imaging, our numerical examples demonstrate that seismic images are generally well approximated by a small percentage of curvelet coefficients, but that not running the optimization algorithm to convergence can harm the reconstruction quality. Similar to large-scale machine learning problems, seismic imaging is computationally too expensive to run optimization algorithms to convergence and typically only a fixed number of data passes (i.e. PDE solves) are possible. In this scenario, the reconstruction is sensitive to the choice of hyper-parameters. Specifically, for the linearized Bregman method, a value of λ that is too large removes too many coefficients from the image, while a value that is too small, allows noise to re-enter the solution. Running the algorithm for a fixed number of data passes therefore does not guarantee that all coefficients such as weak reflectors or diffractors are able to re-enter the solution. Our examples show that these effects are typically less severe in parts of the image with good illumination, but that areas with poor illumination are more sensitive to hyper-parameters. On the other hand, the areas of poor illumination are also the parts of the images that exhibit the strongest subsampling artifacts and where the ben-

efit of sparsity-promotion is the most apparent. Generally, making exact predictions about the behavior of the approximation error is challenging, as nonlinear image approximations (i.e. keeping a fixed percentage of sorted coefficients), are not as well understood as linear image approximations.

3.5 Conclusion

Least-squares reverse-time migration in the frequency domain avoids the problem of having to store or recompute a large number of time-domain wavefields for computing gradients with the adjoint-state method. Conventionally, the gradient has to be computed for each frequency separately by solving the corresponding Helmholtz equation. On-the-fly Fourier transforms offer the possibility to compute monochromatic wavefields with a time-domain modeling code and to obtain an arbitrary number of frequencies within a single time-stepping loop. Formulating least-squares migration as a sparsity-promoting minimization problem allows us to work with small random subsets of shots and frequencies, thus making it possible to perform least-squares imaging at a fraction of the cost of conventional LS-RTM, using as few as two passes through the data set and without having to save or recompute time-domain wavefields.

On-the-fly discrete Fourier transforms offer a fast and easy-to-implement alternative to optimal checkpointing, in which the amount of memory and computational overhead does not depend on the number of time steps, but on the number of frequencies for which the gradient is computed. By varying the batch size of shots or frequencies, the method allows to choose a trade-off between the amount of computations and memory versus the number of iterations and the quality of the final result. Optimal checkpointing on the other hand, provides a trade off between memory usage versus the amount of additional computations, as fewer checkpoints require more time steps that need to be recomputed. This makes imaging with on-the-fly Fourier transforms interesting for applications where both storage and CPU time are expensive, such as cloud computing, as we can trade image quality

for computational resources. Another advantageous scenario for our approach is the case where only a small amount of computational resources are available for a long time, in which case we can trade computational resources for a large number of iterations, where each iteration only uses a small number of frequencies and is cheap in terms of both storage and compute.

REFERENCES

[1] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, Nov. 1983.

[2] N. D. Whitmore, "Iterative depth migration by backward time propagation," *53rd Annual International Meeting, SEG, Expanded Abstracts*, pp. 382–385, 1983.

[3] G. Lambare, J. Virieux, R. Madariaga, and S. Jin, "Iterative asymptotic inversion in the acoustic approximation," *Geophysics*, vol. 57, no. 9, pp. 1138–1154, 1992.

[4] G. T. Schuster, "Least-squares cross-well migration," *63th Annual International Meeting, SEG, Expanded Abstracts*, pp. 110–113, 1993.

[5] T. Nemeth, C. Wu, and G. T. Schuster, "Least-squares migration of incomplete reflection data," *Geophysics*, vol. 64, no. 1, pp. 208–221, 1999.

[6] A. A. Valenciano, "Imaging by wave-equation inversion," PhD thesis, Stanford University, 2008.

[7] Y. Tang and B. Biondi, "Least-squares migration/inversion of blended data," *79th Annual International Meeting, SEG, Expanded Abstracts*, pp. 2859–2863, 2009.

[8] S. Dong, J. Cai, M. Guo, S. Suh, Z. Zhang, B. Wang, and Z. Li, "Least-squares reverse time migration: Towards true amplitude imaging and improving the resolution," in *82nd Annual International Meeting, SEG, Expanded Abstracts*, 2012, pp. 1–5.

[9] C. Zeng, S. Dong, and B. Wang, "Least-squares reverse time migration: Inversion-based imaging toward true reflectivity," *The Leading Edge*, vol. 33, pp. 962–964, 966, 968, 9 2014.

[10] F. J. Herrmann, "Randomized sampling and sparsity: Getting more information from fewer samples," *Geophysics*, vol. 75, no. 6, WB173–WB187, 2010.

[11] T. van Leeuwen, A. Y. Aravkin, and F. J. Herrmann, "Seismic Waveform Inversion by Stochastic Optimization," *International Journal of Geophysics*, no. 2011, 2011.

[12] W. Dai, W. Xin, and G. Schuster, "Least-squares migration of multisource data with a deblurring filter," *Geophysics*, vol. 76, no. 5, R135–R146, Sep. 2011.

[13] W. Dai, P. Fowler, and G. Schuster, "Multi-source least-squares reverse time migration," *Geophysical Prospecting*, vol. 70, pp. 681–695, 4 Jun. 2012.

[14] W. Dai, Y. Huang, and G. T. Schuster, "Least-squares reverse time migration of marine data with frequency-selection encoding," *Geophysics*, vol. 78, no. 4, S233–S242, Jul. 2013.

[15] Y. Liu, "Multisource least-squares extended reverse time migration with preconditioning guided gradient method," *83rd Annual International Meeting, SEG, Expanded Abstracts*, pp. 3709–3715, 2013.

[16] W. Huang and H.-W. Zhou, "Stochastic conjugate gradient method for least-square seismic inversion problems," in *84th Annual International Meeting, SEG, Expanded Abstracts*. 2014, pp. 4003–4007.

[17] C. Li, J. Huang, Z. Li, and R. Wang, "Plane-wave least-squares reverse time migration with a preconditioned stochastic conjugate gradient method," *Geophysics*, vol. 83, no. 1, S33–S46, 2018.

[18] Y. Chen, J. Yuan, S. Zu, S. Qu, and S. Gan, "Seismic imaging of simultaneous-source data using constrained least-squares reverse time migration," *Journal of Applied Geophysics*, vol. 114, pp. 32–35, Mar. 2015.

[19] F. J. Herrmann and X. Li, "Efficient least-squares imaging with sparsity promotion and compressive sensing," *Geophysical Prospecting*, vol. 60, pp. 696–712, 2012.

[20] X. Lu, L. Han, J. Yu, and X. Chen, "L1 norm constrained migration of blended data with the FISTA algorithm," *Journal of Geophysics and Engineering*, vol. 12, pp. 620–628, 2015.

[21] N. Tu and F. Herrmann, "Fast imaging with surface-related multiples by sparse inversion," *Geophysical Journal International*, vol. 201, no. 1, pp. 304–417, 2015.

[22] E. Candès, L. Demanet, D. Donoho, and L. Ying, "Fast discrete Curvelet transforms," *Multiscale Modeling & Simulation*, vol. 5, no. 3, pp. 861–899, 2006.

[23] F. J. Herrmann, P. P. Moghaddam, and C. Stolk, "Sparsity- and continuity- promoting seismic image recovery with Curvelet frames," *Applied and Computational Harmonic Analysis*, vol. 24, pp. 150–173, 2008.

[24] S. Fomel and Y. Liu, "Seislet transform and seislet frame," *Geophysics*, vol. 75, no. 3, pp. V25–V38, 2010.

[25] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, no. 8, p. 1259, 1984.

[26] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[27] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[28] W. W. Symes, "Reverse time migration with optimal checkpointing," *Geophysics*, vol. 72, no. 5, SM213–SM221, 2007.

[29] G. A. McMechan, "Migration by extrapolation of time-dependent boundary values," *Geophysical Prospecting*, vol. 31, no. 3, pp. 413–420, 1983.

[30] B. D. Nguyen and G. A. McMechan, "Five ways to avoid storing source wavefield snapshots in 2D elastic prestack reverse time migration," *Geophysics*, vol. 80, no. 1, S1–S18, 2015.

[31] C. M. Furse, "Faster than Fourier-ultra-efficient time-to-frequency domain conversions for FDTD," in *Institute of Electrical and Electronics Engineers (IEEE) Antennas and Propagation Society International Symposium*, vol. 1, Jun. 1998, 536–539 vol.1.

[32] K. T. Nihei and X. Li, "Frequency response modelling of seismic waves using finite difference time domain with phase sensitive detection," *Geophysical Journal International*, vol. 169, no. 3, pp. 1069–1078, 2007.

[33] K. Watanabe, "Green's functions for Laplace and wave equations," in *Integral Transform Techniques for Green's Function*. Cham: Springer International Publishing, 2015, pp. 33–76, ISBN: 978-3-319-17455-6.

[34] L. Sirgue, J. Etgen, U. Albertin, and S. Brandsberg-Dahl, *System and method for 3D frequency domain waveform inversion based on 3D time-domain forward modeling*, US Patent 7,725,266, May 2010.

[35] V. Etienne, S. Operto, J. Virieux, Y. Jia, *et al.*, "Computational issues and strategies related to full waveform inversion in 3D elastic media: Methodological developments," in *80th Annual International Meeting, SEG, Expanded Abstracts*, Society of Exploration Geophysicists, 2010, pp. 1050–1054.

[36] Y. Kim, C. Shin, H. Calandra, and D.-J. Min, "An algorithm for 3D acoustic time-Laplace-Fourier-domain hybrid full waveform inversion," *Geophysics*, vol. 78, no. 4, R151, 2013.

[37] K. Xu and G. A. McMechan, "2D frequency-domain elastic full-waveform inversion using time-domain modeling and a multistep-length gradient approach," *Geophysics*, vol. 79, no. 2, R41–R53, 2014.

[38] W. Ha, S.-G. Kang, and C. Shin, "3D Laplace-domain waveform inversion using a low-frequency time-domain modeling algorithm," *Geophysics*, vol. 80, no. 1, R1–R13, 2015.

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

[40] L. Sirgue and R. G. Pratt, "Efficient waveform inversion and imaging: A strategy for selecting temporal frequencies," *Geophysics*, vol. 69, no. 1, pp. 231–248, 2004.

[41] F. J. Herrmann, N. Tu, and E. Esser, "Fast "online" migration with Compressive Sensing," *77th Conference and Exhibition, EAGE, Expanded Abstracts*, 2015.

[42] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, "Devito: Towards a generic finite difference DSL using symbolic Python," in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, IEEE, 2016, pp. 67–75.

[43] M. Louboutin, P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 1: Forward modeling," *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, 2017.

[44] R. G. Pratt, "Seismic waveform inversion in the frequency domain, Part 1: Theory and verification in a physical scale model," *Geophysics*, vol. 64, no. 3, pp. 888–901, 1999.

[45] K. Yoon and K. J. Marfurt, "Reverse-time migration using the Poynting vector," *Exploration Geophysics*, vol. 37, no. 1, pp. 102–107, 2006.

[46] A. Guitton, B. Kaelin, and B. Biondi, "Least-squares attenuation of reverse-time-migration artifacts," *Geophysics*, vol. 72, no. 1, S19–S23, 2007.

[47] H. Zhu, Y. Luo, T. Nissen-Meyer, C. Morency, and J. Tromp, "Elastic imaging and time-lapse migration based on adjoint methods," *Geophysics*, vol. 74, no. 6, WCA167–WCA177, 2009.

[48] W. Zhou, R. Brossier, S. Operto, and J. Virieux, "Full waveform inversion of diving & reflected waves for velocity model building with impedance inversion based on scale separation," *Geophysical Journal International*, vol. 202, no. 3, pp. 1535–1554, 2015.

[49] T. J. Op't Root, C. C. Stolk, and M. V. de Hoop, "Linearized inverse scattering based on seismic reverse time migration," *Journal de Mathematiques Pures et Appliquees*, vol. 98, no. 2, pp. 211–238, 2012.

[50] N. D. Whitmore and S. Crawley, "Applications of RTM inverse scattering imaging conditions," *82nd Annual International Meeting, SEG, Expanded Abstracts*, pp. 1–6, 2012.

[51] P. A. Witte, M. Yang, and F. J. Herrmann, "Sparsity-promoting least-squares migration with the linearized inverse scattering imaging condition," *79th Conference and Exhibition, EAGE, Expanded Abstracts*, 2017.

[52] D. Miller, M. Oristaglio, and G. Beylkin, "A new slant on seismic imaging: Migration and integral geometry," *Geophysics*, vol. 52, no. 7, pp. 943–964, Jul. 1987.

[53] W. A. Mulder and R.-E. Plessix, "How to choose a subset of frequencies in frequency-domain finite-difference migration," *Geophysical Journal International*, vol. 158, no. 3, pp. 801–812, 2004.

[54] D. Donoho, "Compressed sensing," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[55] E. J. Candès, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006.

[56] N. Tu, A. Y. Aravkin, T. van Leeuwen, and F. J. Herrmann, "Fast least-squares migration with multiples and source estimation," *75th Conference and Exhibition, EAGE, Expanded Abstracts*, 2013.

[57] L. A. Romero, D. C. Ghiglia, C. C. Ober, and S. A. Morton, "Phase encoding of shot records in prestack migration," *Geophysics*, vol. 65, no. 2, pp. 426–436, 2000.

[58] A. Buades, B. Coll, and J.-M. Morel, "A review of image denoising algorithms, with a new one," *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.

[59] J. Ma and G. Plonka, "The Curvelet Transform," *Institute of Electrical and Electronics Engineers (IEEE), Signal Processing Magazine*, vol. 27, no. 2, pp. 118–133, Mar. 2010.

[60] G. Dutta, "Sparse least-squares reverse time migration using Seislets," *Journal of Applied Geophysics*, vol. 136, pp. 142–155, 2017.

[61] D. A. Lorenz, F. Schöpfer, and S. Wenger, "The Linearized Bregman method via Split Feasibility Problems: Analysis and Generalizations," *Society for Industrial and*

Applied Mathematics (SIAM) Journal on Imaging Sciences, vol. 7, pp. 1237–1262, 2 2014.

[62] W. Yin, "Analysis and Generalizations of the Linearized Bregman Method," *Society for Industrial and Applied Mathematics (SIAM) Journal on Imaging Sciences*, vol. 3, no. 4, pp. 856–877, 2010.

[63] D. A. Lorenz, S. Wenger, F. Schöpfer, and M. Magnor, "A sparse Kaczmarz solver and a Linearized Bregman method for online compressed sensing," *Institute of Electrical and Electronics Engineers (IEEE): International Conference on Image Processing*, pp. 1347–1351, 2014.

[64] H. Mansour and Ö. Yilmaz, *A fast randomized Kaczmarz algorithm for sparse solutions of consistent linear systems*, `http://arxiv.org/abs/1305.3803`, Computing Research Reposityory (arXiv CoRR, 2013. (visited on 11/19/2015).

[65] P. A. Witte, M. Louboutin, and F. J. Herrmann, *The Julia Devito Inversion Framework (JUDI)*, `https://github.com/slimgroup/JUDI.jl`, Apr. 2019. (visited on 04/15/2019).

[66] R. Clayton and B. Engquist, "Absorbing boundary conditions for acoustic and elastic wave equations," *Bulletin of the Seismological Society of America*, vol. 67, no. 6, pp. 1529–1540, 1977.

[67] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, *High-level Python abstractions for optimal checkpointing in inversion problems*, `https://arxiv.org/abs/1802.02474`, Computing Research Repository (arXiv CoRR), Jan. 2018. (visited on 02/14/2018).

[68] E. Bergsma, *Sigsbee2a 2D synthetic dataset*, `http://www.delphi.tudelft.nl/SMAART/sigsbee2a.htm`, Publicly Released 'SMAART' Data Sets, Sep. 2001. (visited on 02/21/2018).

[69] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *International Business Machines Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, Mar. 1967.

[70] F. Billette and S. Brandsberg-Dahl, "The 2004 BP velocity benchmark.," in *67th Annual International Meeting, EAGE, Expanded Abstracts*, EAGE, 2005, B035.

[71] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[72] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. A. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, *Architecture and performance of Devito, a system for automated stencil computation*, `https://arxiv.org/abs/1807.03032`, Computing Research Repository (arXiv CoRR), 2018. (visited on 07/21/2018).

Part III

Seismic imaging in the cloud

CHAPTER 4

AN EVENT-DRIVEN APPROACH TO SEISMIC IMAGING IN THE CLOUD

4.1 Introduction

Seismic imaging of the earth's subsurface is one of the most computationally expensive applications in scientific computing, as state-of-the-art imaging methods such as least-squares reverse time migration (LS-RTM), require repeatedly solving a large number of forward and adjoint wave equations during numerical optimization [e.g. 1, 2, 3, 4]. Similar to training neural networks, the gradient computations in seismic imaging are based on backpropagation and require storage or re-computations of the state variables (i.e. of the forward modeled wavefields). Due to the large computational cost of repeatedly modeling wave propagation over many time steps using finite difference modeling, seismic imaging requires access to high-performance computing (HPC) clusters, but the high cost of acquiring and maintaining HPC cluster makes this option only viable for a small number of major energy companies [5, 6]. For this reason, cloud computing has lately emerged as a possible alternative to on-premise HPC clusters, offering many advantages such as no upfront costs, a pay-as-you-go pricing model and theoretically unlimited scalability. Outside of the HPC community, cloud computing is today widely used by many companies for general purpose computing, data storage and analysis or machine learning. Customers of cloud providers include major companies such as General Electric (GE), Comcast, Shell or Netflix, with the latter hosting their video streaming content on Amazon Web Services (AWS) [7]. Netflix' utilization of the cloud for large-scale video streaming has acted as a driver for improving the scalability of cloud tools such as object storage and event-driven computations [8], which are not available on conventional HPC environments and which we will subsequently adapt for our purposes.

However, adapting the cloud for high-performance computing applications such as seismic imaging, is not straight-forward, as numerous investigations and case studies have shown that the cloud generally cannot provide the same performance, low latency, high bandwidth and mean time between failures (MTBF) as conventional HPC clusters. An early performance analysis by Jackson [9] of a range of typical NERSC HPC applications on Amazon's Elastic Compute Cloud (EC2) found that, at the time of the comparison, applications on EC2 ran between 2.7 to 50 times slower than on a comparable HPC system due to poor network performance and that the latency was up to 400 times worse. A performance analysis by [10] using standard benchmark suites such as the HPC challenge (HPCC) supports these observations, finding that the performance on various cloud providers is in the order of one magnitude worse than to comparable HPC clusters. Other performance studies using standardized benchmarks suites, as well as domain-specific applications, similarly conclude that poor network performance severely limits the HPC capabilities of the cloud [11, 12, 13, 14, 15, 16, 17].

While communication and reliability are the strongest limiting factors in the performance of HPC applications in the cloud, several investigations [18, 19, 20] point out that embarrassingly parallel applications show in fact very good performance that is comparable to (non-virtualized) HPC environments. Similarly, performance tests on single cloud nodes and bare-metal instances using HPCC and high-performance LINPACK benchmarks [21], demonstrate good performance and scalability as well [22, 23]. These findings underline that the *lift and shift approach* for porting HPC applications to the cloud is unfavorable, as most HPC codes are based on highly synchronized message passing (i.e. MPI) [24] and rely on stable and fast network connections, which are not (yet) available. On the other hand, compute nodes and architectures offered by cloud computing are indeed comparable to current supercomputing systems [23] and the cloud offers a range of novel technologies such as cloud object storage or event-driven computations [25]. These technologies are not available on traditional HPC systems and make it possible to address computational bottle-

necks of HPC in fundamentally new ways. Cloud object storage, such as Amazon's Simple Storage Service (S3) [26] or Google Cloud Storage [27], are based on the distribution of files to physically separated data centers and thus provide virtual unlimited scalability, as the storage system is not constrained by the size and network capacity of a fixed number of servers [28]. Successfully porting HPC applications to the cloud therefore requires a careful re-architecture of the corresponding codes and software stacks to take advantage of these technologies, while minimizing communication and idle times. This process is heavily application dependent and requires the identification of how a specific application can take advantage of specialized cloud services such as serverless compute or high throughput batch processing to mitigate resilience issues, avoid idle instances and thus minimize cost.

Based on these premises, we present a workflow for large-scale seismic imaging on AWS, which does not rely on a conventional cluster of virtual machines, but is instead based on a serverless visual workflow that takes advantage of the mathematical properties of the seismic imaging optimization problem [29]. Similar to deep learning, objective functions in seismic imaging consist of a sum of (convex) misfit functions and iterations of the associated optimization algorithms exhibit the structure of a MapReduce program [30]. The map part corresponds to computing the gradient of each element in the sum and is embarrassingly parallel to compute, but individual gradient computations are expensive as they involve solving partial differential equations (PDEs). The reduce part corresponds to the summation of the gradients and update of the model parameters and is comparatively cheap to compute, but I/O intensive. Instead of performing these steps on a cluster of permanently running compute instances, our workflow is based on specialized AWS services such as AWS Batch and Lambda functions, which are responsible for automatically launching and terminating the required computational resources [31, 25]. EC2 instances are only running as long as they are utilized and are shut down automatically as soon as computations are finished, thus preventing instances from sitting idle. This stands in contrast to alternative MapReduce cloud services, such as Amazon's Elastic Map Reduce (EMR),

which is based on Apache Hadoop and relies on a cluster of permanently running EC2 instances [32, 33]. In our approach, expensive gradient computations are carried out by AWS Batch, a service for processing embarrassingly parallel workloads, but with the possibility of using (MPI-based) domain decomposition for individual solutions of partial differential equations (PDEs). The cheaper gradient summations are performed by Lambda functions, a service for serverless computations, in which code is run in response to events, without the need to manually provision computational resources [25].

The following section provides an overview of the mathematical problem that underlies seismic imaging and we identify possible characteristics that can be taken advantage of to avoid the aforementioned shortcomings of the cloud. In the subsequent section, we describe our seismic imaging workflow, whose initial version has been developed for AWS, but the underlying services are available on other cloud platforms as well (i.e. Google Compute Cloud, Azure). We then present a performance analysis of our workflow on a real-world seismic imaging application, using a popular subsurface benchmark model [34]. Apart from conventional scaling tests, we also consider specific cloud metrics such as resilience and cost, which, aside from the pure performance aspects like scaling and time-to-solution, are important practical considerations for HPC in the cloud. The final section reports on our experience of porting our workflow to Azure and we present a large-scale seismic imaging case study in 3D, thus highlighting the cross-platform portability of our approach.

4.2 Problem Overview

Seismic imaging and parameter estimation are a set of computationally challenging inverse problems with high practical importance, as they are today widely used in the oil and gas (O&G) industry for geophysical exploration, as well as for monitoring geohazards. In the context of exploration, seismic imaging can significantly increase the success rate of drilling into reservoirs, thus reducing both cost and environmental impact of resource exploration [35].

Mathematically, seismic imaging and parameter estimation are PDE-constrained optimization problems, that are typically expressed in the following (unconstrained) form [36, 37]:

$$\underset{\mathbf{m}}{\text{minimize}} \;\; \Phi(\mathbf{m}) = \sum_{i=1}^{n_s} \frac{1}{2} ||\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i||_2^2, \tag{4.1}$$

where $\mathcal{F}(\mathbf{m}, \mathbf{q}_i)$ represents the solution of the acoustic wave equation for a given set of model parameters $\mathbf{m}$. The evaluation of this operator corresponds to modeling seismic data for a given subsurface model (or image) $\mathbf{m}$ and a known source function $\mathbf{q}_i$ by solving a wave equation using time-domain finite-difference modeling. The vector $\mathbf{d}_i$ denotes the observed seismic measurements at the i^{th} location of the seismic source, which is moved along the surface within the survey area (Figure 4.1). In essence, the goal of seismic inversion is to find a set of model parameters $\mathbf{m}$, such that the numerically modeled data matches the observed data from the seismic survey. The total number of individual source experiments n_s for realistic surveys, i.e. the number of PDEs that have to solved for each evaluation of $\Phi(\mathbf{m})$, is quite large and lies in the range of 10^3 for 2D surveys and 10^5 for 3D surveys.

Seismic inverse problems of this form are typically solved with gradient-based optimization algorithms such as (stochastic) gradient descent, (Gauss-) Newton methods, sparsity-promoting minimization or constrained optimization [e.g. 38, 39] and therefore involve computing the gradient of equation 4.1 for all or a subset of indices i. The gradient of the objective function is given by:

$$\mathbf{g} = \sum_{i=1}^{n_s} \mathbf{J}^\top \Big(\mathcal{F}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i \Big), \tag{4.2}$$

where the linear operator $\mathbf{J} = \frac{\partial \mathcal{F}(\mathbf{m}, \mathbf{q}_i)}{\partial \mathbf{m}}$ is the partial derivative of the forward modeling operator with respect to the model parameters $\mathbf{m}$ and $\top$ denotes the matrix transpose. Both the objective function, as well as the gradient exhibit a sum structure over the source indices

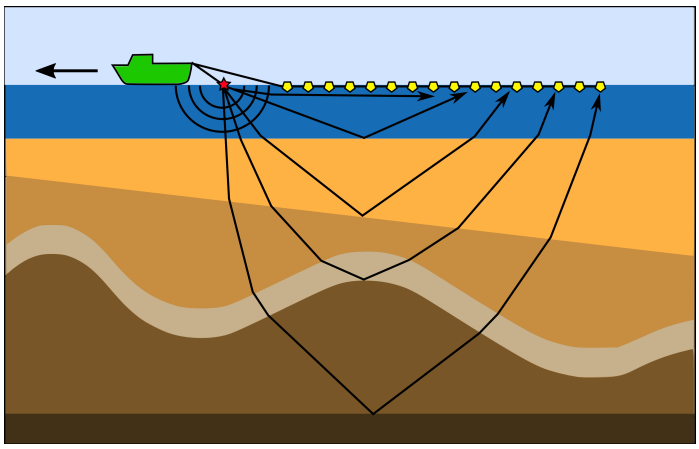


Figure 4.1: A two-dimensional depiction of marine seismic data acquisition. A vessel fires a seismic source and excites acoustic waves that travel through the subsurface. Waves are reflected and refracted at geological interfaces and travel back to the surface, where they are recorded by an array of seismic receivers that are towed behind the vessel. The receivers measure pressure changes in the water as a function of time and receiver number for approximately 10 seconds, after which the process is repeated. A typical seismic survey consists of several thousand of these individual source experiments, during which the vessel moves across the survey area.

and are embarrassingly parallel to compute. Evaluating the objective function and computing the gradient are therefore instances of a MapReduce program [30], as they involve the parallel computation and subsequent summation of elements of the sum. However, computing the gradient for a single index i involves solving two PDEs, namely a forward wave equation and an adjoint (linearized) wave equation (denoted as a multiplication with $\mathbf{J}^\top$). For realistically sized 3D problems, the discretized model in which wave propagation is modeled has up to 10^9 variables and modeling has to be performed for several thousand time steps. The number of time steps is determined by the time stepping interval and depends on the wave speed and the temporal frequency of the data and increases significantly as these properties change [40]. The observed seismic data $\mathbf{d}_i$ ($i = 1, ..., n_s$) is typically in the range of several terabytes and a single element of the data (a seismic *shot record*) ranges from several mega- to gigabytes.

The problem structure of equation 4.1 is very similar to deep learning and the parallels between convolutional neural networks and PDEs have lately attracted strong attention

[41]. As in deep learning, computing the gradient of the objective function (equation 4.2) is based on backpropagation and in principle requires storing the state variables of the forward problem. However, in any realistic setting the wavefields are too big to be stored in memory and therefore need to be written to secondary storage devices or recomputed from a subset of checkpoints [42]. Alternatively, domain decomposition can be used to reduce the domain size per compute node such that the forward wavefields fit in memory, or time-to frequency conversion methods can be employed to compute gradients in the frequency domain [43, 4]. In either case, computing the gradient for a given index i is expensive both in terms of necessary floating point operations, memory and IO and requires highly optimized finite-difference modeling codes for solving the underlying wave equations. Typical computation times of a single (3D-domain) gradient $\mathbf{g}_i$ (i.e. one element of the sum) are in the range of minutes to hours, depending on the domain size and the complexity of the wave simulator, and the computations have to be carried out for a large number of source locations and iterations.

The high computational cost of seismic modeling, in combination with the complexity of implementing optimization algorithms to solve equation 4.1, leads to enormously complex inversion codes, which have to run efficiently on large-scale HPC clusters. A large amount of effort goes into implementing fast and scalable wave equation solvers [44, 45, 46], as well as into frameworks for solving the associated inverse problem [47, 48, 49, 50]. Codes for seismic inversion are typically based on message passing and use MPI to parallelize the loop of the source indices (equation 4.1). Furthermore, a nested parallelization is oftentimes used to apply domain-decomposition or multi-threading to individual PDE solves. The reliance of seismic inversion codes on MPI to implement an embarrassingly parallel loop is disadvantageous in the cloud, where the mean-time-between failures (MTBF) is much shorter than on HPC systems [9] and instances using spot pricing can be arbitrarily shut down at any given time [51]. Another important aspect is that the computation time of individual gradients can vary significantly and cause load imbalances and large

idle times, which is problematic in the cloud, where users are billed for running instances by the second, regardless of whether the instances are in use or idle. For these reasons, we present an alternative approach for seismic imaging in the cloud based on batch processing and event-driven computations.

4.3 Event-driven seismic imaging on AWS

4.3.1 Workflow

Optimization algorithms for minimizing equation 4.1 essentially consists of three steps. First, the elements of the gradient $\mathbf{g}_i$ are computed in parallel for all or a subset of indices $i \in n_s$, which corresponds to the map part of a MapReduce program. The number of indices for which the objective is evaluated defines the batch size of the gradient. The subsequent reduce part consists of summing these elements into a single array and using them to update the unknown model/image according to the rule of the respective optimization algorithm (Algorithm 4.1). Optimization algorithms that fit into this general framework include variations of stochastic/full gradient descent (GD), such as Nesterov's accelerated GD [52] or Adam [53], as well as the nonlinear conjugate gradient method [54], projected GD or iterative soft thresholding [55]. Conventionally, these algorithms are implemented as a single program and the gradient computations for seismic imaging are parallelized using message passing. Running MPI-based programs of this structure in the cloud requires that users request a set of EC2 instances and establish a network connection between all workers [56]. Tools like StarCluster [57] or AWS HPC [58] facilitate the process of setting up a cluster and even allow adding or removing instances to a running cluster. However, adding or dropping instances during the execution of an MPI program is not easily possible, so the number of instances has to stay constant during the entire length of the program execution, which, in the case of seismic inversion, can range from several days to weeks. This makes this approach not only prone to resilience issues, but it can result in significant cost overhead, if workloads are unevenly distributed and instances are temporarily idle.

Algorithm 4.1 Generic algorithm structure for gradient-based minimization of equation 4.1, using a fixed number of iterations n. Many gradient-based algorithms exhibit this overall MapReduce structure, including stochastic/mini-batch gradient descent (GD) with various update rules (e.g. Nesterov, Adam), projected GD, iterative soft thresholding or the nonlinear conjugate gradient method.

1: Input: batch size n_b, max. number of iterations n, step size α, initial guess $\mathbf{m}_1$

2: **for** $i = 1$ to n **do**

3: Compute gradients $\mathbf{g}_i$, $i = 1, ..., n_b$ in parallel

4: Sum gradients: $\mathbf{g} = \sum_{i=1}^{n_b} \mathbf{g}_i$

5: Update optimization variable, e.g. using SGD:

$$\mathbf{m}_{k+1} = \mathbf{m}_k - \alpha \mathbf{g}$$

6: **end for**

Instead of implementing and running optimization algorithms for seismic inverse problems as a single program that runs on a cluster of EC2 instances, we express the steps of a generic optimization algorithm through AWS Step Functions (Figure 4.2) and deploy its individual components through a range of specialized AWS services [59]. Step functions allow the description of an algorithm as a collection of states and their relationship to each other using the JavaScript Object Notation (JSON). From the JSON definition of a workflow, AWS renders an interactive visual workflow in the web browser, as shown in Figure 4.2. For our purpose, we use Step Functions to implement our iterative loop [60], during which we compute and sum the gradients, and use them to update the seismic image. We choose Step Functions to express our algorithm, as they allow composing different AWS Services such as AWS Batch and Lambda functions into a single workflow, thus making it possible to leverage preexisiting AWS services and to combine them into a single application. Another important aspect of Step Functions is that the execution of the workflow itself is managed by AWS and does not require running any EC2 instances, which is why we refer to this approach as *serverless*. During execution time, AWS automatically progresses the workflow from one state to the next and users are only billed for

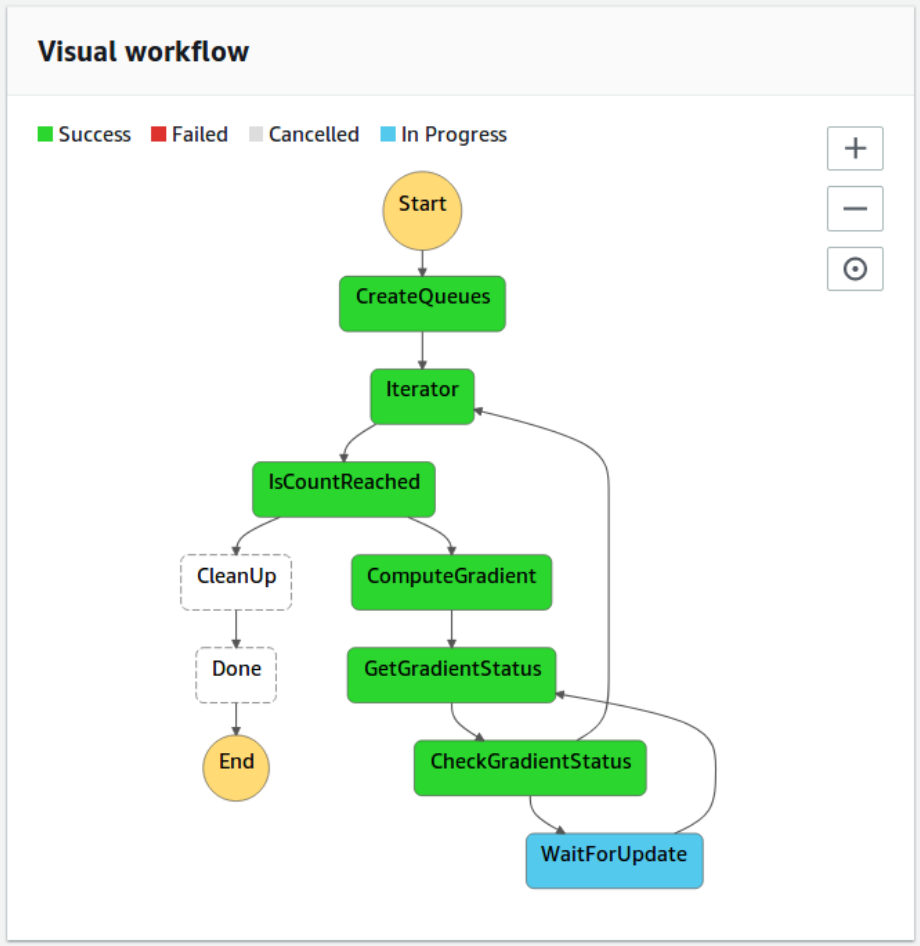


Figure 4.2: A generic seismic imaging algorithm, expressed as a serverless visual workflow using AWS Step Functions. The workflow consists as a collection of states, which are used to implement an iterative optimization loop. Each iteration involves computing the gradient of equation 4.1 using AWS Batch, as well an updating the optimization variable (i.e. the seismic image).

transitions between states, but the cost is negligible compared to the cost of running EC2 instances (0.025\$ per $1,000$ state transitions) [59].

States can be simple if-statements such as the `IsCountReached` state, which keeps track of the iteration number and terminates the workflow after a specified number of iterations, but states can also be used invoke other AWS services. Specifically, states can be used to invoke AWS Lambda functions to carry out serverless computations. Lambda functions allow users to run code in response to events, such as invocations through AWS Step Functions, and automatically assign the required amount of computational resources

to run the code. Billing is based on the execution time of the code and the amount of used memory. Compared to EC2 instances, Lambda functions have a much shorter startup time in the range of milliseconds rather than minutes, but they are limited to 3 GB of memory and an execution time of 15 minutes. As such, Lambda functions themselves are not suitable for carrying out the gradient computations, but they can be used to manage other AWS services. In our workflow, we use Lambda functions invoked by the `ComputeGradient` state (Figure 4.2) to launch AWS Batch jobs for computing the gradients. During the gradient computation, which can take up to several hours, the Step Functions check in a user-defined interval if the full gradient has been computed, before advancing the workflow to the next state. The `WaitForGradient` state pauses the workflow for a specified amount of time, during which no additional computational resources are running other than the AWS Batch job itself.

4.3.2 Computing the gradient

The gradient computations (equation 4.2) are the major workload of seismic inversion, as they involve solving forward and adjoint wave equations, but the embarrassingly parallel structure of the problem lends itself to high-throughput batch computing. On AWS, embarrassingly parallel workloads can be processed with AWS Batch, a service for scheduling and running parallel containerized workloads on EC2 instances [31]. Parallel workloads, such as computing a gradient of a given batch size, are submitted to a batch queue and AWS Batch automatically launches the required EC2 instances to process the workload from the queue. Each job from the queue runs on an individual instance or set of instances, with no communication being possible between individual jobs.

In our workflow, we use the Lambda function invoked by the `ComputeGradient` state (Figure 4.2) to submit the gradient computations to an AWS Batch queue. Each element of the gradient $\mathbf{g}_i$ corresponds to an individual job in the queue and is run by AWS Batch as a separate Docker container [61]. Every container computes the gradient for its

respective source index i and writes its resulting gradient to an S3 bucket (Figure 4.3), Amazon's cloud object storage system [26]. The gradients computed by our workflow are one-dimensional numpy arrays of the size of the vectorized seismic image and are stored in S3 as so-called objects [62]. Once an individual gradient $\mathbf{g}_i$ has been computed, the underlying EC2 instance is shut down automatically by AWS Batch, thus preventing EC2 instances from idling. Since no communication between jobs is possible, the summation of the individual gradients is implemented separately using AWS Lambda functions. For this purpose, each jobs also sends its S3 object identifier to a message queue (SQS) [63], which automatically invokes the reduction stage (Figure 4.4). For the gradient computations, each worker has to download the observed seismic data of its respective source index from S3 and the resulting gradient has to be uploaded to S3 as well. The bandwidth with which objects are up- and downloaded is only limited by the network bandwidth of the EC2 instances and ranges from 10 to 100 Gbps [64]. Notably, cloud object storage such as S3 has no limit regarding the number of workers that can simultaneously read and write objects, as data is (redundantly) distributed among physically separated data centers, thus providing essentially unlimited IO scalability [26].

AWS Batch runs jobs from its queue as separate containers on a set of EC2 instances, so the source code of the application has to be prepared as a Docker container. Containerization facilitates portability and has the advantage that users have full control over managing dependencies and packages. Our Docker image contains the code for solving acoustic wave equations to compute gradients of a respective seismic source location. Since this is the most computational intensive part of our workflow, it is important that the wave equation solver is optimized for performance, but is also implemented in a programming language that allows interfacing other AWS services such as S3 or SQS. In our workflow, we use the domain-specific language compiler called Devito for implementing and solving the underlying wave equations using time-domain finite-difference modeling [46, 65]. Devito is implemented in Python and provides an application programming interface (API)

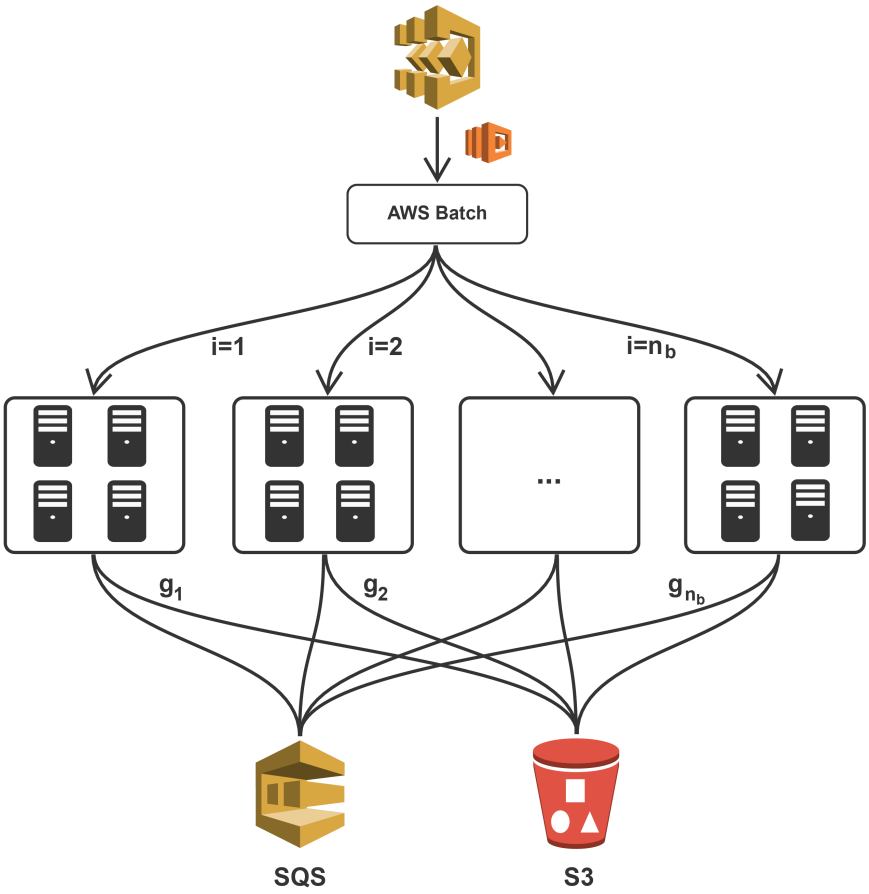


Figure 4.3: The gradients of the LS-RTM objective function are computed as an embarrassingly parallel workload using AWS Batch. This process is automatically invoked by the AWS Step Functions (Figure 4.2) during each iteration of the workflow. The gradients of individual source locations are computed as separate jobs on either a single or multiple EC2 instances. Communication is only possible between instances of a single job, but not between separate jobs. The resulting gradients are saved in S3 and the respective object names are sent to an SQS queue to invoke the gradient summation.

for implementing forward and adjoint wave equations as high-level symbolic expressions based on the SymPy package [66]. During runtime, the Devito compiler applies a series of performance optimizations to the symbolic operators, such as reductions of the operation count, loop transformations, and introduction of parallelism [65]. Devito then generates optimized finite-difference stencil code in C from the symbolic Python expressions and dynamically compiles and runs it. Devito supports both multi-threading using OpenMP, as well as generating code for MPI-based domain decomposition. Its high-level API allows expressing wave equations of arbitrary stencil orders or various physical representations

without having to implement and optimize low-level stencil codes by hand. Furthermore, Devito includes various possibilities for backpropagation, such as optimal checkpointing or on-the-fly Fourier transforms [67, 68]. The complexity of implementing highly optimized and parallel wave equation solvers is therefore abstracted and vertically integrated into the AWS workflow.

By default, AWS Batch runs the container of each job on a single EC2 instance, but recently AWS introduced the possibility to run multi-node batch computing jobs [69]. Thus, individual jobs from the queue can be computed on a cluster of EC2 instances and the corresponding Docker containers can communicate via the AWS network. As for single-instance jobs, AWS Batch automatically requests and terminates the EC2 instances on which the Docker containers are deployed. In the context of seismic imaging and inversion, multi-node batch jobs enable nested levels of parallelization, as we can use AWS Batch to parallelize the sum of the source indices, while using MPI-based domain decomposition and/or multi-threading for solving the underlying wave equations. This provides a large amount of flexibility in regard of the computational strategy for performing backpropagation and how to address the storage of the state variables. AWS Batch allows to scale horizontally, by increasing the number of EC2 instances of multi-node jobs, but also enables vertical scaling by adding additional cores and/or memory to single instances. In our performance analysis, we compare and evaluate different strategies for computing gradients with Devito regarding scaling, costs and turnaround time.

4.3.3 Gradient reduction

Every computed gradient is written by its respective container to an S3 bucket, as no communication between individual jobs is possible. Even if all gradients in the job queue are computed by AWS Batch in parallel at the same time, we found that the computation time of individual gradients typically varies considerably (up to 10 percent), due to varying network performance or instance capacity. Furthermore, we found that the startup time of the

underlying EC2 instances itself is highly variable as well, so jobs in the queue are usually not all started at the same time. Gradients therefore arrive in the bucket over a large time interval during the batch job. For the gradient reduction step, i.e. the summation of all gradients into a single array, we take advantage of the varying time-to-solutions by implementing an event-driven gradient summation using Lambda functions. In this approach, the gradient summation is not performed by as single worker or the master process who has to wait until all gradients have been computed, but instead summations are carried out by Lambda functions in response to gradients being written to S3. The event-driven summation is therefore started as soon as the first two gradients have been computed.

The event-driven gradient summation is automatically invoked through SQS messages, which are sent by the AWS Batch workers that have completed their computations and have saved their respective gradient to S3. Before being shut down, every batch worker sends a message with the corresponding S3 object name to an AWS SQS queue, in which all object names are collected (Figure 4.4). Sending messages to SQS invokes AWS Lambda functions that read up to 10 messages at a time from the queue. Every invoked Lambda function that contains at least two messages, i.e. two object names, reads the corresponding arrays from S3, sums them into a single array, and writes the array as a new object back to S3. The new object name is sent to the SQS queue, while the previous objects and objects names are removed from the queue and S3. The process is repeated recursively until all n_b gradients have been summed into a single array, with n_b being the batch size for which the gradient is computed. The gradient summation is implemented in Python, which is one of the languages supported by AWS Lambda [25]. SQS guarantees that all messages are delivered at least once to the subscribing Lambda functions, thus ensuring that no gradients are lost in the summation process [63].

Since Lambda functions are limited to 3 GB of memory, it is not always possible to read the full gradient objects from S3. Gradients that exceed Lambda's available memory are therefore streamed from S3 using appropriate buffer sizes and are re-uploaded to S3 using

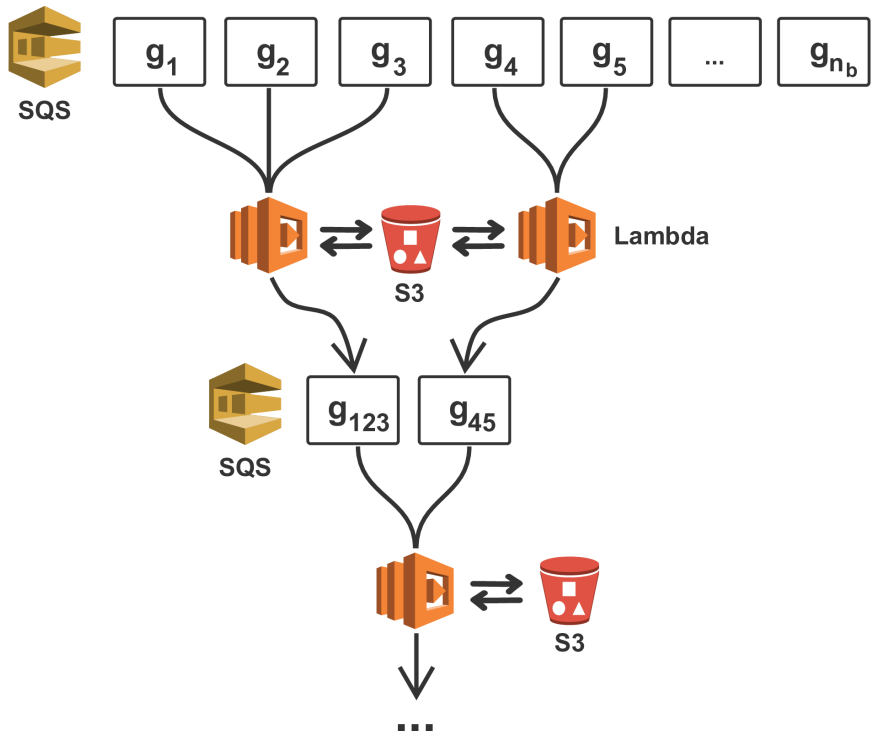


Figure 4.4: Event-driven gradient summation using AWS Lambda functions. An SQS message queue collects the object names of all gradients that are currently stored in S3 and automatically invokes Lambda functions that stream up to 10 files from S3. Each Lambda function sums the respective gradients, writes the result back to S3 and sends the new object name to the SQS queue. The process is repeated until all gradients have been summed into a single S3 object. SQS has a guaranteed at-least-once delivery of messages to ensure that no objects are lost in the summation.

the `multipart_upload` functions of the S3 Python interface [70]. As the execution time of Lambda functions is furthermore limited to 15 minutes, the bandwidth of S3 is not sufficient to stream and re-upload objects that exceed a certain size within a single Lambda invocation. For this case, we include the possibility that the workers of the AWS Batch job split the computed gradients into smaller chunks that are saved separately in S3, with the respective objects names being sent to multiple SQS queues. The gradient summation is then performed in chunks by separate queues and Lambda functions. The `CreateQueues` task of our Step Functions workflow (Figure 4.2) automatically creates the required number of queues before starting the optimization loop and the `CleanUp` state removes them after the final iteration.

The advantage of the event-based gradient reduction is that that the summation is executed asynchronously, as soon as at least two S3 objects containing gradients are available, while other batch jobs are still running. Therefore, by the time the last batch worker finishes the computation of its respective gradient, all remaining gradients have already been summed into a single object, or at least a small number of objects. Furthermore, summing files of a single queue happens in parallel (if enough messages are in the queue), as multiple Lambda functions can be invoked at the same time. Furthermore, splitting the gradients itself into chunks that are processed by separate queues leads to an additional layer of parallelism. In comparison to a fixed cluster of EC2 instances, the event-driven gradient summation using Lambda function also takes advantage of the fact that the summation of arrays is computationally considerably cheaper than solving wave equations and therefore does not require to be carried out on the expensive EC2 instances used for the PDE solves.

4.3.4 Variable update

Once the gradients have been computed and summed into a single array that is stored as an S3 object, the gradient is used to update the optimization variables of equation 4.1, i.e. the seismic image or subsurface parameters such as velocity. Depending on the specific objective function and optimization algorithm, this can range from simple operations like multiplications with a scalars (gradient descent) to more computational expensive operations such as sparsity promotion or applying constraints [71]. Updates that use entry-wise operations only and are cheap to compute such as multiplications with scalars or soft-thresholding, can be applied directly by the Lambda functions in the final step of the gradient summation. I.e. the Lambda function that sums the final two gradients, also streams the optimization variable of the current iteration from S3, uses the gradient to update it and directly writes the updated variable back to S3.

Many algorithms require access to the full optimization variable and gradient, such as Quasi-Newton methods and other algorithms that need to compute gradient norms. In

this case, the variable update is too expensive and memory intensive to be carried out by Lambda functions and has to be submitted to AWS Batch as a single job, which is then executed on a larger EC2 instance. This can be accomplished by adding an extra state such as `UpdateVariable` to our Step Functions workflow. However, to keep matters simple, we only consider a simple stochastic gradient descent example with a fixed step size in our performance analysis, which is computed by the Lambda functions after summing the final two gradients [72]. The `CheckGradientStatus` state of our AWS Step Functions advances the workflow to the next iteration, once the updated image (or summed gradient) has been written to S3. The workflow shown in Figure 4.2 terminates the optimization loop after a predefined number of iterations (i.e. epochs), but other termination criteria based on gradient norms or function values are possible too and can be realized by modifying the `IsCountReached` state. The update of the optimization variable concludes a single iteration of our workflow, whose performance we will now analyze in the subsequent sections.

4.4 Performance analysis

In our performance analysis, we are interested in the performance of our workflow on a real-world seismic imaging application regarding scalability, cost and turn-around time, as well as the computational benefits and overhead introduced by our event-driven approach. We conduct our analysis on a popular 2D subsurface velocity model (Figure 4.5), called the 2004 BP velocity estimation benchmark model [34]. This model was originally created for analyzing seismic processing or inversion algorithms, but as the model represents a typical large-scale 2D workload, we consider this model for our following performance analysis. The seismic data set of this model contains $1,348$ seismic source locations and corresponding observations $\mathbf{d}_i$ ($i = 1, ..., 1,348$). The (unknown) seismic image has dimensions of $1,911 \times 10,789$ grid points, i.e. a total of almost 21 million parameters. An overview of all grid parameters and data dimensions are presented in Table A.2.

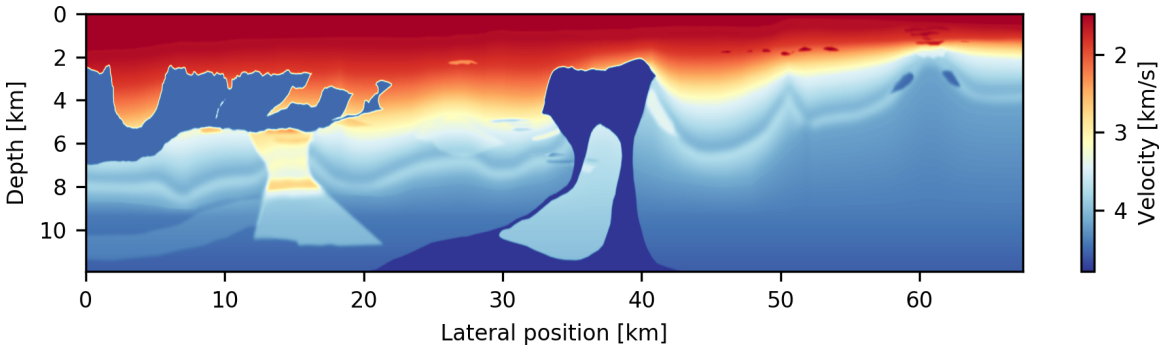


Figure 4.5: The BP 2004 benchmark model, a 2D subsurface velocity model for development and testing of algorithms for seismic imaging and parameter estimation [34]. This model and the corresponding seismic data set are used in our performance analysis. The velocity model and the unknown image have dimensions of $1,911 \times 10,789$ grid points, a total of 20.1 million unknown parameters.

4.4.1 Weak scaling

In our first performance test, we analyze the weak scaling behavior of our workflow by varying the batch size (i.e. the number of source locations) for which the gradient of the LS-RTM objective function (equation 4.1) is computed. For this test, we perform a single iteration of stochastic gradient descent (SGD) using our workflow and measure the time-to-solution as a function of the batch size. The workload per instance, i.e. per parallel worker, is fixed to one gradient. The total workload for a specified batch size is submitted to AWS Batch as a so-called *array job*, where each array entry corresponds to a single gradient $\mathbf{g}_i$. AWS Batch launches one EC2 instance per array entry (i.e. per gradient), runs the respective container on the instance and then terminates the instance afterwards. If a sufficient amount of instances are available, AWS Batch will theoretically launch all containers of the array job instantaneously and run the full workload in parallel, but as we will see in the experiment, this is in practice not necessarily the case.

In the experiment, we measure the time-to-solution for performing a single iteration of our workflow, i.e. one SGD update. We exclude the setup time of the SQS queues, which is the first step of our workflow (Figure 4.2), as this process only has to be performed once, prior to the first iteration. Therefore, each run involves the following steps:

1. A Lambda function submits the AWS Batch job for specified batch size n_b (Figure 4.3)

2. Compute gradients $\mathbf{g}_i$ ($i = 1, ..., n_b$) in parallel (Figure 4.3)

3. Lambda functions sum the gradients (Figure 4.4):
   $\mathbf{g} = \sum_{i=1}^{n_b} \mathbf{g}_i$

4. A Lambda function performs the SGD update of the image: $\mathbf{x} = \mathbf{x} - \alpha\mathbf{g}$

We define the time-to-solution as the the time interval between the submission of the AWS Batch job by a Lambda function (step 1) and the time stamp of the S3 object containing the updated image (step 4). This time interval represents a complete iteration of our workflow.

The computations of the gradients are performed on `m4.4xlarge` instances and the number of threads per instance is fixed to 8, which is the number of physical cores that is available on the instance. The `m4` instance is a general purpose EC2 instance and we chose the instance size (`4xlarge`) such that we are able to store the wavefields for backpropagation in memory. The workload for each batch worker consists of solving a forward wave equation to model the predicted seismic data and an adjoint wave equation to backpropagate the data residual and to compute the gradient. For this and all remaining experiments, we use the acoustic isotropic wave equation with a second order finite difference (FD) discretization in time and 8th order in space. We model wave propagation for 12 seconds, which is the recording length of the seismic data. The time stepping interval is given by the Courant-Friedrichs-Lewy condition with 0.55 ms, resulting in $21,889$ time steps. Since it is not possible for the waves to propagate through the whole domain within this time interval, we restrict the modeling grid to a size of $1,911 \times 4,001$ grid points around the current source location. After modeling, each gradient is extended back to the full model size ($1,911 \times 10,789$ grid points). A detailed description of the setup parameters and utilized software and hardware is provided in the appendix (Table A.1). The dimensions of

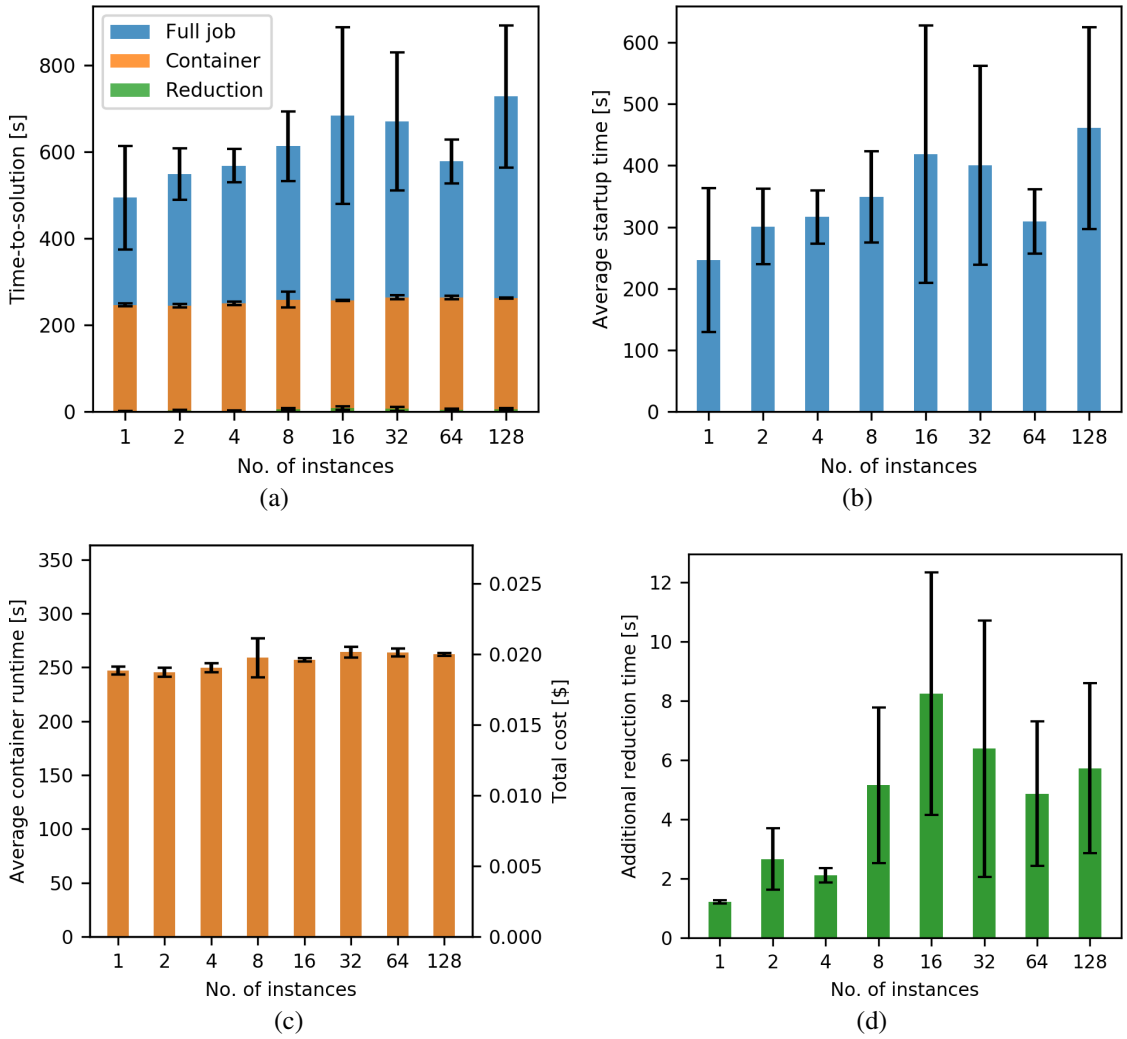


Figure 4.6: Weak scaling results for performing a single iteration of stochastic gradient as a function of the batch size for which the gradient is computed (a). The gradient is computed as an AWS Batch job with an increasing number of parallel EC2 instances, while the gradient summation and the variable update are performed by Lambda functions. The total time-to-solution (a) consists of the average time it takes AWS Batch to request and start the EC2 instances (b), the average runtime of the containers (c) and the additional reduction time (d), i.e. the time difference between the final gradient of the respective batch and the updated image. All timings are the arithmetic mean over three runs, with the error bars representing the standard deviation.

this example represent a large-scale 2D example, but all components of our workflow are agnostic to the number of physical dimensions and are implemented for three-dimensional domains as well. The decision to limit the examples to a 2D model was purely made from a financial viewpoint and to make the results reproducible in a reasonable amount of time.

The timings ranging from a batch size of 1 to 128 are displayed in Figure 4.6a. The batch size corresponds to the number of parallel EC2 instances on which the jobs are executed. The time-to-solution consists of three components that make up the full runtime of each job:

1. The average time for AWS Batch to request and launch the EC2 instances and to start the Docker containers on those instances.

2. The runtime of the containers

3. The additional gradient reduction and image update time, which is given by the time interval between the termination of the AWS Batch job and the time stamp of the updated variable.

The sum of these components makes up the time-to-solution as shown in Figure 4.6a and each component is furthermore plotted separately in Figures 4.6b to 4.6d. All timings are the arithmetic mean over three individual runs and the standard deviation is indicated by the error bars. The container runtimes of Figure 4.6c are the arithmetic mean of the individual container runtimes on each instance (varying from 1 to 128). The average container runtime is proportional to the cost of computing one individual gradient and is given by the container runtime times the price of the `m4.4xlarge` instance, which was $0.2748 per hour at the time of testing. AWS Batch automatically launches and terminates the EC2 instance on which each gradient is computed and the user only pays for utilized EC2 time. No extra charges occurs for AWS Batch itself, i.e. for scheduling and launching the batch job.

The timings indicate that the time-to-solution generally grows as the batch size, and therefore the number of containers per job, increases (Figure 4.6a). A close up inspection of the individual components that make up the total time-to-solution shows that this is mostly due to the increase of the startup time, i.e. the average time it takes AWS Batch to schedule and launch the EC2 instances of each job (Figure 4.6b). We monitored the status

of the EC2 instances during the job execution and found that AWS Batch does generally not start all instances of the array job at the same time, but instead in several stages, over the course of 1 to 3 minutes. The exact startup time depends on the batch size and therefore on the number of instances that need to be launched, but also on the availability of the instance within the AWS region. The combination of these factors lead to an increase of the average startup time for an increasing batch size, but also to a large variance of the startup time between individual runs. Unfortunately, the user has no control over the startup time, but it is important to consider that no cost is incurred during this time period, as no EC2 instances are running while the individual containers remain in the queue.

The average container runtime, i.e. the average computation time of a single gradient within the batch, is fairly stable as the batch size increases (Figure 4.6c). This observation is consistent with the fact that each container of an AWS Batch array job runs as an individual Docker container and is therefore independent of the batch size. The container runtime increases only slightly for larger batch sizes and we observe a large variance in some of the container runtimes (specifically for a batch size of 8). This variance stems from the fact that users do not have exclusive access to the EC2 instances on which the containers are deployed. Specifically, our containers run on `m4.4xlarge` instances, which have 8 cores (16 virtual CPUs) and 64 GB of memory. In practice, AWS deploys these instances on larger physical nodes and multiple EC2 instances (of various users) can run on the same node. We hypothesize that a larger batch size increases the chance of containers being deployed to a compute node that runs at full capacity, thus slightly increasing the average container runtime, as user do not have exclusive access to the full network capacity or memory bandwidth.

Finally, we also observe an increase in the additional gradient reduction time, i.e. the interval between the S3 timestamps of the final computed gradient $\mathbf{g}_i$ and the updated image $\mathbf{x}$. The batch size corresponds to the number of gradients that have to be summed before the gradient can be used to update the image.The event-driven gradient reduction

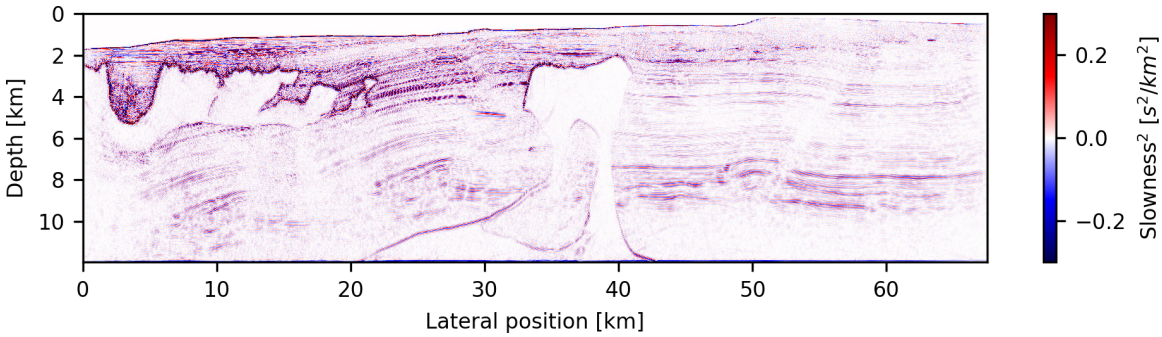


Figure 4.7: Final seismic image after 30 iterations of stochastic gradient descent and a batch size of 80, which corresponds to approximately two passes through the data set (i.e. two epochs).

invokes the summation process as soon as the first gradients are written to S3, so most gradients are already summed by the time the final worker finishes its gradient computation. For the event-driven gradient summation, the variance of the startup and container runtime is therefore advantageous, as it allows the summation to happen asynchronously. However, in our example, the time interval between the first two gradients being written to S3 (thus invoking the gradient reduction) and the final gradient being computed, does not appear to be large enough to complete the summation of all gradients. Specifically, we see an increase in the reduction time from a batch size of 4 to 8, after which the additional reduction is mostly constant, but again with a large variance. This variance is due to a non-deterministic component of our event-based gradient summation, resulting from a limitation of AWS Lambda. While users can specify a maximum number of messages that Lambda functions read from an SQS queue, it is not possible to force Lambda to read a minimum amount of two messages, resulting in most Lambda functions reading only a single message (i.e. one object name) from the queue. Since we need at least two messages to sum the corresponding gradients, we return the message to the queue and wait for a Lambda invocation with more than one message. The user has no control over this process and sometimes it takes several attempts until a Lambda function with multiple messages is invoked. The likelihood of this happening increases with a growing batch size, since a larger number of gradients need to be summed, which explains the increase of the reduction

time and variance in Figure 4.6d.

Overall, the gradient summation and variable update finishes within a few seconds after the last gradient is computed and the additional reduction time is small compared to the full time-to-solution and to the pure computation time of the gradients. In our example, the startup time (Figure 4.6b) takes up the majority of the time-to-solution (Figure 4.6a), as it lies in the range of a few minutes and is in fact longer than the average container runtime of each worker (Figure 4.6c). However, the startup time is independent of the runtime of the containers, so the ratio of the startup time to the container runtime improves as the workload per container increases. Furthermore, the cost of the batch job only depends on the container runtime and the batch size, but not on the startup time or reduction time. The cost for summing the gradients is given by the cumulative runtime of the Lambda functions, but is negligible compared to the EC2 cost for computing the gradients. At the time of the example, the cost for Lambda functions was $\$2 \cdot 10^{-7}$ per request and $\$1.6 \cdot 10^{-5}$ per used GB-second. Figure 4.7 shows the final seismic image that is obtained after running our workflow for 30 iterations and a batch size of 80, which corresponds to 1.8 epochs. The source locations in each iteration are chosen from a uniform random distribution and and after the final iteration, each data sample (i.e. seismic shot record) has been, in expectation, used 1.8 times. In this example, every gradient was computed by AWS Batch on a single instance and a fixed number of threads, but in the subsequent section we analyze the scaling of runtime and cost as a function of the number of cores and EC2 instances. Furthermore, we will analyze in a subsequent example how the cost of running the gradient computations with AWS Batch compares to performing those computations on a fixed cluster of EC2 instances.

4.4.2 Strong scaling

In the following set of experiments, we analyze the strong scaling behavior of our workflow for an individual gradient calculation, i.e. a gradient for a batch size of 1. For this,

we consider a single gradient computation using AWS Batch and measure the runtime as a function of either the number of threads or the number of instances in the context of MPI-based domain decomposition. In the first experiment, we evaluate the vertical scaling behavior, i.e. we run the gradient computation on a single instance and vary the number of OpenMP threads. In contrast to the weak scaling experiment, we model wave propagation in the full domain ($1,911 \times 10,789$ grid points), to ensure that the sub-domain of each worker is not too small when we use maximum number of threads. The measured runtime is the sum of the kernel times spent for solving the forward and the adjoint wave equation and therefore excludes memory allocation time and code generation time.

Since AWS Batch runs all jobs as Docker containers, we compare the runtimes with AWS Batch to running our application on a bare metal instance, in which case we have direct access to the compute node and run our code without any virtualization. All timings on AWS are performed on a `r5.24xlarge` EC2 instance, which is a memory optimized instance type that uses the Intel Xeon Platinum 8175M architecture. The `24xlarge` instance has 96 virtual CPU cores (48 physical cores on 2 sockets) and 768 GB of memory. Using the largest possible instance of the `r5` class, ensures that our AWS Batch job has exclusive access to the physical compute node. Bare metal instances automatically give users exclusive access to the full node. We also include the Optimum HPC cluster in our comparison, a small research cluster at the University of British Columbia based on the Intel's Ivy Bridge 2.8 GHz E5-2680v2 processor. Optimum has 2 CPUs per node and 10 cores per CPU.

Figure 4.8a shows the comparison of the kernel runtimes on AWS and Optimum and Figure 4.8b displays the corresponding speedups. As expected, the `r5` bare metal instance shows the best scaling, as it uses a newer architecture than Optimum and does not suffer from the virtualization overhead of Docker. We noticed that AWS Batch in its default mode uses hyperthreading (HT), even if we perform thread pinning and instruct AWS Batch to use separate physical cores. As of now, the only way to prevent AWS Batch from performing

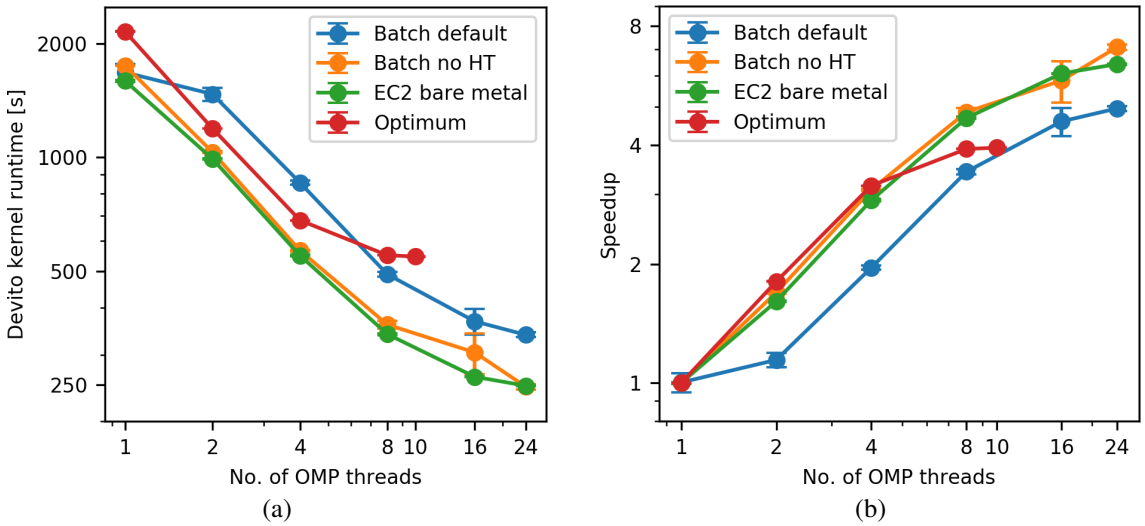


Figure 4.8: Strong scaling results for computing a single image gradient of the BP model as a function of the number of threads. Figure (a) shows the runtimes for AWS Batch with and without hyperthreading, as well as the runtimes on the r5 bare metal instance, in which case no containerization or virtualization is used. For reference, we also provide the runtime on a compute node of an on-premise HPC cluster. Figure (b) shows the corresponding speeds ups.

HT, is to modify the Amazon Machine Image (AMI) of the corresponding AWS compute environment and set the `nr_cpus` parameter of the `/etc/default/grub` file to the number of physical cores per socket (i.e. 24). With HT disabled, the runtimes and speedups of AWS Batch are very close to the timings on the bare-metal instances, indicating that the overhead of Docker affects the runtimes and scaling of our memory-intensive application only marginally, which matches the findings of [73].

Next, we analyze the horizontal strong scaling behavior of running our application with AWS Batch. Once again, we consider the computation of one single gradient, but this time we vary the number of EC2 instances on which the underlying wave equations are solved. We would like to emphasize that AWS Batch is used differently than in the weak scaling experiment, where AWS Batch was used to parallelize the sum over source locations. Multiple workloads (i.e. gradients) were submitted to AWS Batch as an array job and communication between workers of an array job is not possible. Here, we submit a single workload (i.e. one gradient) as a multi-node AWS Batch job, in which case IP-based

communication between instances is enabled. Since this involves distributed memory parallelism, we use domain decomposition based on message passing (MPI) to solve the wave equations on multiple EC2 instances [74, 75]. The code with the corresponding MPI communication statements is automatically generated by the Devito compiler. Furthermore, we use multi-threading on each individual instance and utilize the maximum number of available cores per socket, which is 24 for the `r5` instance and 18 for the `c5n` instance.

We compare the `r5.24xlarge` instance type from the last section with Amazon's recently introduced `c5n` HPC instance. Communication between AWS instances is generally based on ethernet and the `r5` instances have up to 25 GBps networking performance. The `c5n` instance type uses Intel Xeon Platinum 8142M processors with up to 3.4 Ghz architecture and according to AWS provides up to 100 GBps of network bandwidth. The network is based on AWS' Nitro card and the elastic network adapter, but AWS has not disclosed whether this technology is based on InfiniBand or Ethernet [76]. Figures 4.9a and 4.9b show the kernel runtimes and the corresponding speedups ranging from 1 instance to 16 instances. The `r5` instance has overall shorter runtimes than the `c5n` instance, since the former has 24 physical cores per CPU socket, while the `c5n` instance has 18. However, as expected, the `c5n` instance type exhibits a better speedup than the `r5` instance, due to the better network performance. Overall, the observed speed up on both instances types is excellent, with the `c5n` instance archiving a maximum speedup of 11.3 and the `r5` instance of 7.2.

The timings given in Figure 4.9a are once again the pure kernel times for solving the PDEs, but a breakdown of the components that make up the total time-to-solution on the `c5n` instance is provided in Figure 4.9c. The job runtime is defined as the interval between the time stamp at which the batch job was created and the S3 time stamp of the computed gradient. As in our weak scaling test, this includes the time for AWS Batch to request and launch the EC2 instances and to start the Docker containers, but excludes the gradient summation time, since we are only considering the computation of a single gradient. The

container runtime is the runtime of the Docker container on the master node and includes the time it takes AWS Batch to launch the remaining workers and to establish an `ssh` connection between all instances/containers. Currently, AWS Batch requires this process to be managed by the user using a shell script that is run inside each container. After a connection to all workers has been established, the containers run the application as a Python program on each worker. The Python runtime in Figure 4.9c is defined as the runtime of Python on the main node and includes reading the seismic data from S3, allocating memory and Devito's code generation. Our timings in Figure 4.9c show that the overhead from requesting instances and establishing a cluster, i.e. the difference between the Python and container runtime, is reasonable for a small number of instances (less than 2 minutes), but grows significantly as the number instances is increased to 8 and 16. Depending on the runtime of the application, the overhead thus takes up a significant amount of the time-to-solution. In our example, this was the case for 8 and 16 instances, but for more compute-heavy applications that run for one or multiple hours, this amount of overhead may still be acceptable.

Figure 4.9d shows the cost for running our scaling test as a function of the cluster size. The cost is calculated as the instance price (per second) times the runtime of the container on the main node times the number of instances. The cost per gradient grows significantly with the number of instances, as the overhead from establishing an `ssh` connection to all workers increases with the cluster size. The communication overhead during domain decomposition adds an additional layer of overhead that further increases the cost for an increasing number of instances. This is an important consideration for HPC in the cloud, as the shortest time-to-solution does not necessarily correspond to the cheapest approach. Another important aspect is that AWS Batch multi-node jobs do not support spot instances [75]. Spot instances allow users to access unused EC2 capacities at significantly lower price than at the on-demand price, but AWS can terminate spot instances at any time with a two minute warning, e.g. if the demand for that instance type increases [51]. Spot instances are typically in the range of 2 to 3 times cheaper than the corresponding on-demand price,

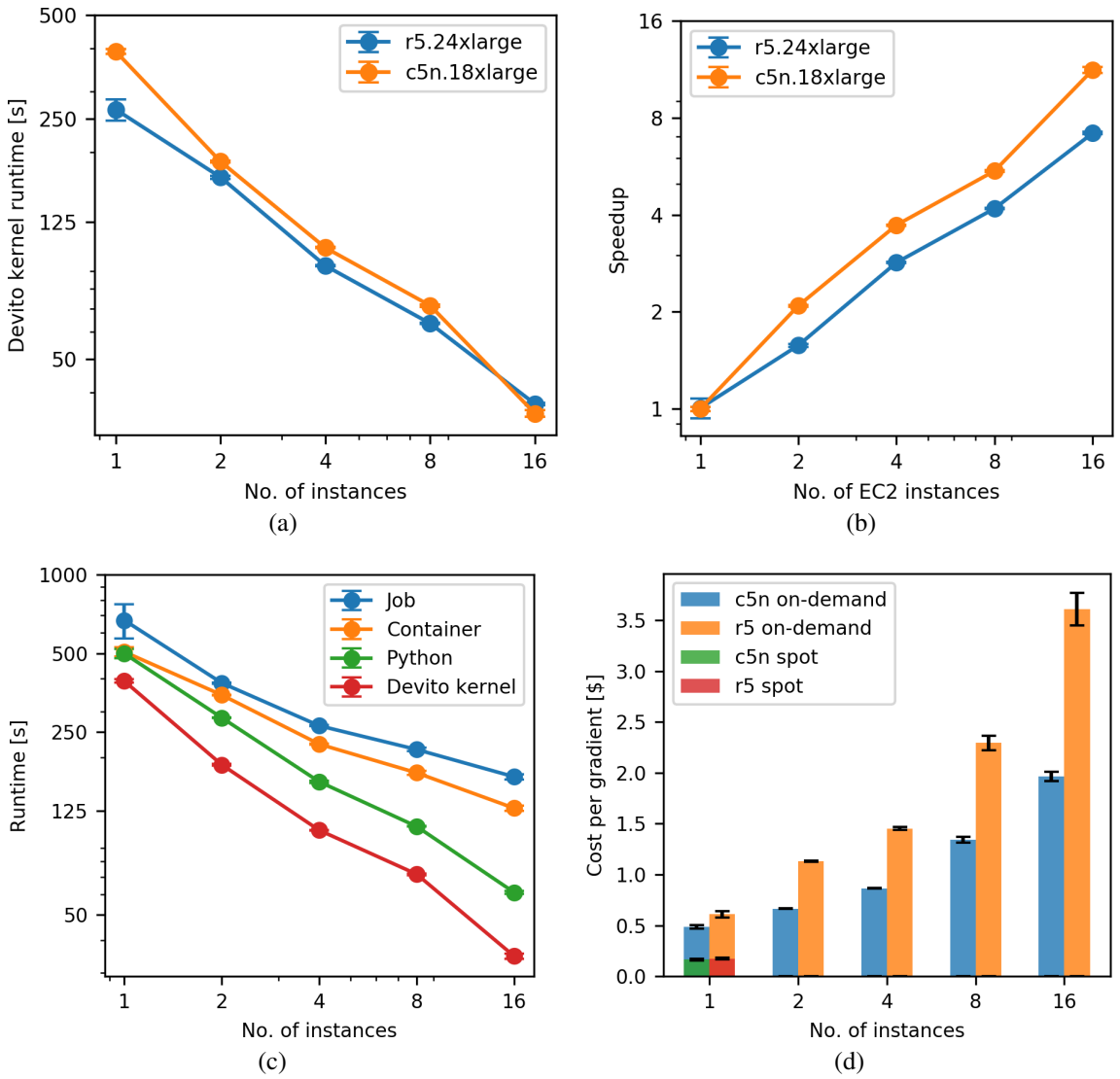


Figure 4.9: Strong scaling results for computing a single gradient as an AWS Batch multi-node job for an increasing number of instances. Figures (a) and (b) show the Devito kernel times and speedups on two different instance types. The observed speedups are 11.3 for the `c5n` and 7.2 for the `r5` instance. Figure (c) shows a breakdown of the time-to-solution of each batch job into its individual components. Figure (d) shows the EC2 cost for computing the gradients. The spot price is only provided for the single-instance batch jobs, as spot instances are not supported for multi-node batch jobs.

but AWS Batch multi-node jobs are, for the time being, only supported with on-demand instances.

The scaling and cost analysis in Figures 4.9a – 4.9d was carried out on the largest instances of the respective instance types (`r5.24xlarge` and `c5n.18xlarge`) to guaran-

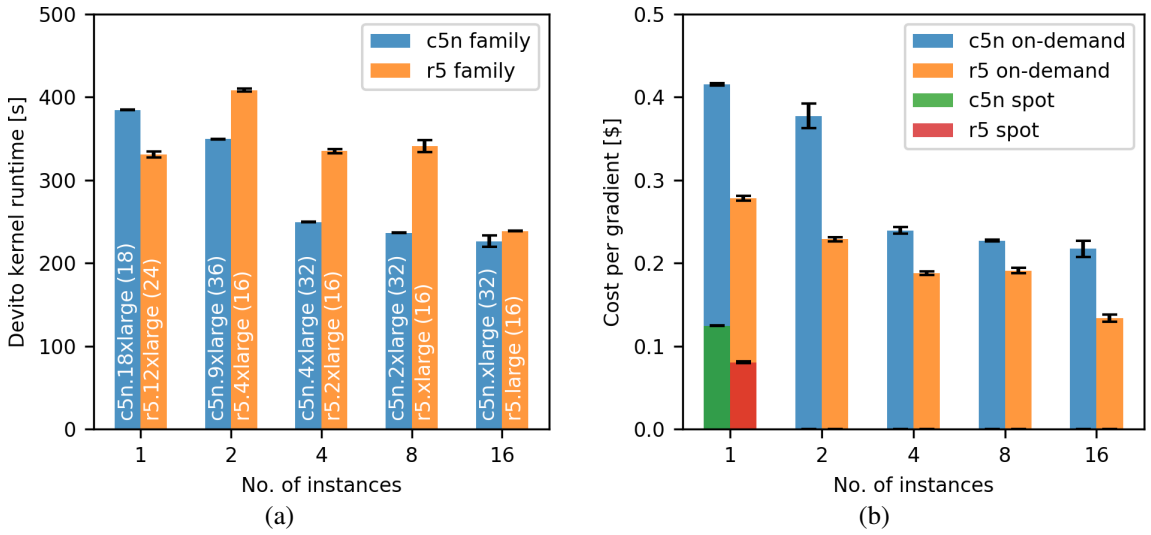


Figure 4.10: Devito kernel runtimes for computing a single gradient as an AWS Batch job for an increasing number of instances. In comparison to the previous example, we use the smallest possible instance type for each job, as specified in each bar. We use the maximum number of available cores on every instance type and the total number of cores across all instances is given in each bar. Figure (b) shows the corresponding cost for computing the gradients. Choosing the instance size such the total memory is approximately constant, avoids that the cost increases as a larger number of instances are used per gradient. However, single instances using spot prices ultimately remain the cheapest option.

tee exclusive access to the compute nodes and network bandwidth. Increasing the number of instances per run therefore not only increases the total number of available cores, but also the amount of memory. However, for computing a single gradient, the required amount of memory is fixed, so increasing the number of instances reduces the required amount of memory per instance, as wavefields are distributed among more workers. In practice, it therefore makes sense to chose the instance type based on the required amount of memory per worker, as memory is generally more expensive than compute. In our specific case, computing the gradient requires 170 GB of memory, which requires either a single `r5.12xlarge` instance, two `r5.4xlarge`, four `r5.2xlarge`, eight `r5.xlarge` or sixteen `r5.large` instances. However, these instances not only differ in the amount of memory, but also in the number of CPU cores. We repeat our previous scaling test, but rather than using the same instance type in all runs, we choose the instance type based on

Table 4.1: Comparison of parallelization strategies on a single EC2 instance in the context of AWS Batch. The timings are the Devito kernel times for computing a single gradient of the BP model using AWS Batch. The program runs as a single docker container on an individual EC2 instance, using either multi-threading (OpenMP) or a hybrid approach of multithreading and domain-decomposition (OpenMP + MPI).

Grid	CPU (cores)	Parallelization	Runtime [s]
$1,911 \times 5,394$	1 (24)	OMP	190.17 ± 7.12
$1,911 \times 10,789$	1 (24)	OMP	378.94 ± 13.57
$1,911 \times 10,789$	2 (48)	OMP	315.92 ± 16.50
$1,911 \times 10,789$	2 (48)	OMP + MPI	249.13 ± 5.22

the required amount of memory. Furthermore, for every instance type, we utilize the maximum amount of available cores using multi-threading with OpenMP. The kernel runtimes for an increasing number of instances is shown in Figure 4.10a. In each bar, we indicate which instance type was used, as well as the total number of cores across all instances. The corresponding costs for computing each gradient is shown in Figure 4.10a. Compared to the previous example, we observe that using 16 small on-demand instances leads to a lower cost than using a single more expensive large instance, but that using a single instance ultimately remains the most cost-effective way of computing a gradient, due to the possibility to utilize spot instances.

In terms of cost, our scaling examples underline the importance of choosing the EC2 instances for the AWS Batch jobs based on the total amount of required memory, rather than based on the amount of CPU cores. Scaling horizontally by using an increasingly large number of instances expectedly leads to a faster time-to-solution, but results in a significant increase of cost as well (Figure 4.9d). As shown in Figure 4.10b, this increase in cost can be avoided to some extent by choosing the instance size such that the total amount of memory stays approximately constant, but ultimately the restriction of not supporting spot instances, makes multi-node batch jobs not attractive in scenarios where single instances provide sufficient memory to run a given application. In practice, it makes therefore sense to use single node/instance batch jobs and to utilize the full number of available cores on each

instance. The largest EC2 instances of each type (e.g. `r5.24xlarge`, `c5n.18xlarge`) have two CPU sockets with shared memory, making it possible to run parallel programs using either pure multi-threading or a hybrid MPI-OpenMP approach. In the latter case, programs still run as a single Docker container, but within each container use MPI for communication between CPU sockets, while OpenMP is used for multithreading on each CPU. For our example, we found that computing a single gradient of the BP model with the hybrid MPI-OpenMP approach leads to a 20 % speedup over the pure OpenMP program (Table 4.1), which correspondingly leads to 20 % cost savings as well.

4.4.3 Cost comparison

One of the most important considerations of high performance computing in the cloud is the aspect of cost. As users are billed for running EC2 instances by the second, it is important to use instances efficiently and to avoid idle resources. This is oftentimes challenging when running jobs on a conventional cluster. In our specific application, gradients for different seismic source locations are computed by a pool of parallel workers, but as discussed earlier, computations do not necessarily complete at the same time. On a conventional cluster, programs with a MapReduce structure, such as parallel gradient computations, are implemented based on a client-server model, in which the workers (i.e. the clients) compute the gradients in parallel, while the master (the server) collects and sums the results. This means that the process has to wait until all gradients $\mathbf{g}_i$ have been computed, before the gradient can be summed and used to update the image. This inevitably causes workers that finish their computations earlier than others to the sit idle. This is problematic when using a cluster of EC2 instances, where the number of instances are fixed, as users have to pay for idle resources. In contrast, the event-driven approach based on Lambda functions and AWS Batch automatically terminates EC2 instances of workers that have completed their gradient calculation, thus preventing resources from sitting idle. This fundamentally shifts the responsibility of requesting and managing the underlying EC2 instances from the

user to the cloud environment and leads to significant cost savings as demonstrated in the following example.

We illustrate the difference between the event-driven approach and using a fixed cluster of EC2 instances by means of a specific example. We consider our previous example of the BP synthetic model and assume that we want to compute the gradient for a batch size of 100. As in our weak scaling experiment, we restrict the modeling domain to the subset of the model that includes the respective seismic source location, as well as the seismic receivers that record the data. Towards the edge of the model, the modeling domain is smaller, as some receivers lie outside the modeling domain and are therefore omitted. We compute the gradient $\mathbf{g}_i$ for 100 random source locations and record the runtimes (Figure 4.11a). We note that most gradients take around 250 seconds to compute, but that the runtimes vary due to different domain sizes and varying EC2 capacity (similar to the timings in Figure 4.6c). We now model the idle times for computing these gradients on a cluster of EC2 instances as a function of the the number of parallel instances, ranging from 1 instance (fully serial) to 100 instances (fully parallel). For a cluster consisting of a single instance, the cumulative idle time is naturally zero, as the full workload is executed in serial by a single instance. For more than one instance, we model the amount of time that each instance is utilized, assuming that the workloads are assigned dynamically to the available instances. The cumulative idle time t_{idle} is then given as the sum of the differences between the runtime of each individual instance t_i and the instance with the longest runtime:

$$t_{\text{idle}} = \sum_{i=1}^{n_{\text{EC2}}} (\max\{t_i\} - t_i), \tag{4.3}$$

The cumulative idle time as a function of the cluster size n_{EC2} is plotted in Figure 4.11b. We note that the cumulative idle time generally increases with the cluster size, as a larger number of instances sit idle while waiting for the final gradient to be computed. On a cluster with 100 instances each gradient is computed by a separate instance, but all workers have

to wait until the last worker finishes its computation (after approximately 387 seconds). In this case, the varying time-to-solutions of the individual gradients leads to a cumulative idle time of 248 minutes. Compared to the cumulative computation time of all gradients, which is 397 minutes, this introduces an overhead of more than 60 percent, if the gradients are computed on a cluster with 100 instances. The cumulative idle time is directly proportional to the cost for computing the 100 gradients, which is plotted on the right axis of Figure 4.11b. With AWS Batch, the cumulative idle time for computing the 100 gradients is zero, regardless of the number of parallel instances that AWS Batch has access to. Any EC2 instance that is not utilized anymore is automatically shut down by AWS Batch, so no additional cost other than the pure computation time of the gradients is invoked [77].

In practice, it is to be expected that the cost savings of AWS Batch are even greater, as we are not taking the time into account that it takes to start an EC2 cluster of a specified number of instances. In our weak scaling experiments (Figure 4.6a), we found that spinning up a large number of EC2 instances does not happen instantaneously but over a period of several minutes, so starting a cluster of EC2 instances inevitably causes some instances to sit idle while the remaining instances are started. This was also observed for our multi-node AWS Batch job experiment (Figure 4.9c), but in this case the cluster size per gradient is considerably smaller than the size of a single large cluster for computing all gradients. Single-node AWS Batch jobs do not suffer from the variable startup time, as workers that are launched earlier than others instantaneously start their computations, without having to wait for the other instances.

While computing the 100 gradients on an EC2 cluster with a small number of instances results in little cumulative idle time, it increases the overall time-to-solution, as a larger number of gradients have to be sequentially computed on each instance (4.11c). With AWS Batch this trade-off does not exist, as the cumulative idle time, and therefore the cost for computing a fixed workload, does not depend on the number of instances. However, it is to be expected that in practice the time-to-solution is somewhat larger for AWS Batch

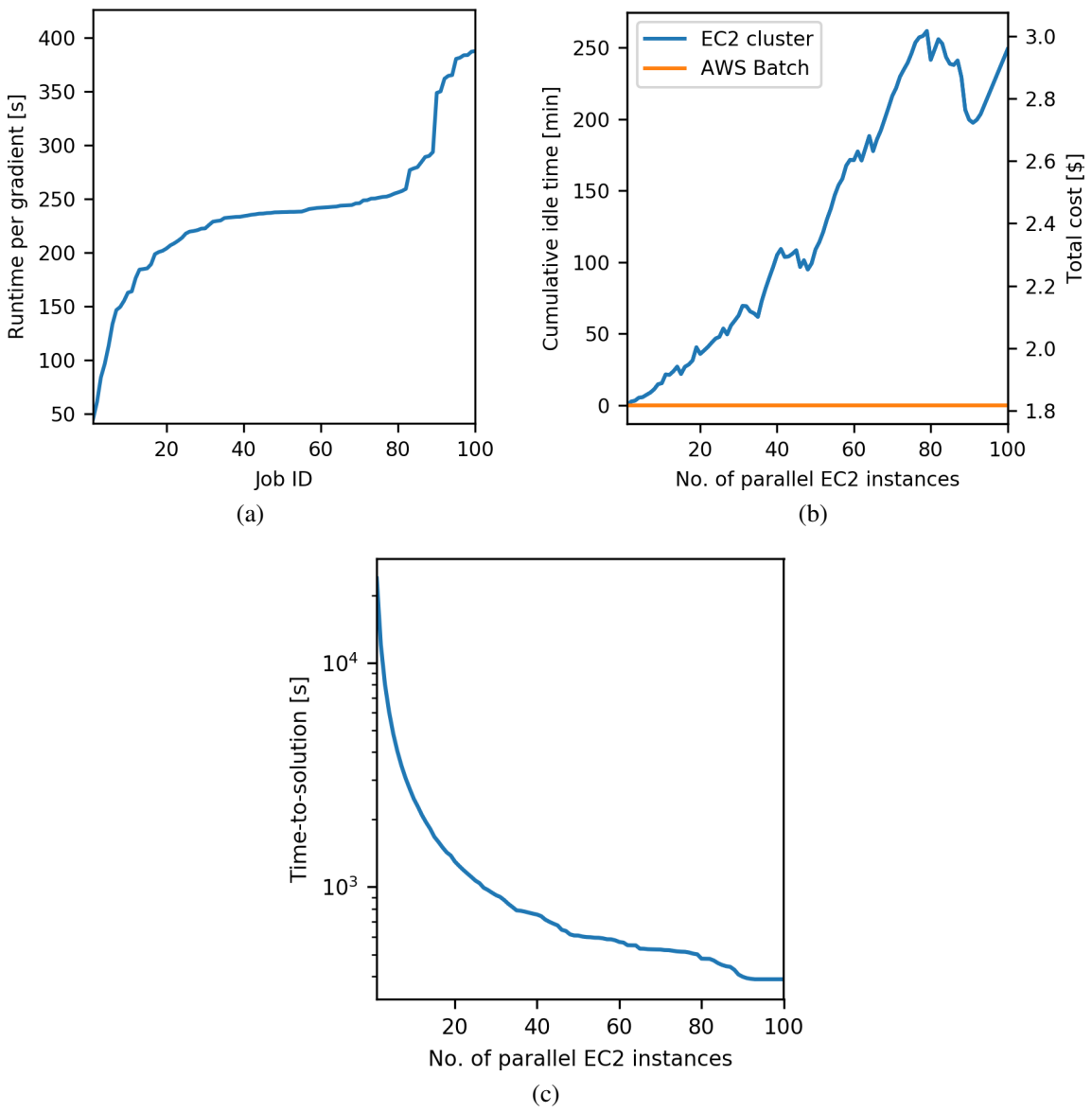


Figure 4.11: (a) Sorted container runtimes of an AWS Batch job in which we compute the gradient of the BP model for a batch size of 100. Figure (b) shows the cumulative idle time for computing this workload as a function of the number of parallel workers on either a fixed cluster of EC2 instances or using AWS Batch. The right-hand y-axis shows the corresponding cost, which is proportional to the idle time. In the optimal case, i.e. no instances every sit idle, the cost for computing a gradient of batch size 100 is 1.8$. Figure (c) shows the time-to-solution as a function of the number of parallel instances, which is the same on an EC2 cluster and for AWS Batch, if we ignore the startup time of the AWS Batch workers or of the corresponding EC2 cluster.

than for a fixed cluster of EC2 instances, as AWS Batch needs to request and launch EC2 instances for every new gradient computation. In our weak scaling experiments in which

Table 4.2: AWS on-demand and spot prices of a selection of EC2 instance types that were used in the previous experiments. Prices are provided for the US East (North Virginia) region, in which all experiments were carried out (07/11/2019).

Instance	On-demand ($/hour)	Spot ($/hour)	Ratio
m4.4xlarge	0.800	0.2821	2.84
r5.24xlarge	6.048	1.7103	3.54
c5n.18xlarge	3.888	1.1659	3.33

we requested up to 128 instances, we found that the corresponding overhead lies in the range of 3 to 10 minutes (per iteration), but it is to be expected that the overhead further grows for an even larger number of instances. However, no additional cost is introduced while AWS Batch waits for the EC2 instances to start.

4.4.4 Cost saving strategies for AWS Batch

Using AWS Batch for computing the gradients of a seismic imaging workflow limits the runtime that each EC2 instance is active to the amount of time it takes to compute a single gradient (of one source location). Running a seismic imaging workflow on a conventional cluster of EC2 instances, requires instances to stay up during the entire execution time of the program, i.e. for all iterations of the seismic imaging optimization algorithm. The limitation of instance runtimes to the duration of a single gradient computations with AWS Batch is beneficial for the usage of spot instances, as it reduces the chance that a specific instance is shut down within the duration it is used. As demonstrated in our earlier examples (Figures 4.9d and 4.10b), spot instances can significantly reduce the cost of running EC2 instances, oftentimes by a factor of $2-4$ in comparison to on-demand instances (Table 4.2).

In contrast to on-demand instances, the price of spot instances is not fixed and depends on the current demand and availability of the instance type that is being requested. As such, prices for spot instances can vary significantly between the different zones of a specific region (Figure 4.12a). If spot instances are used for a cluster and are fixed for the entire duration of the program execution, users are exposed to variations of the spot price during

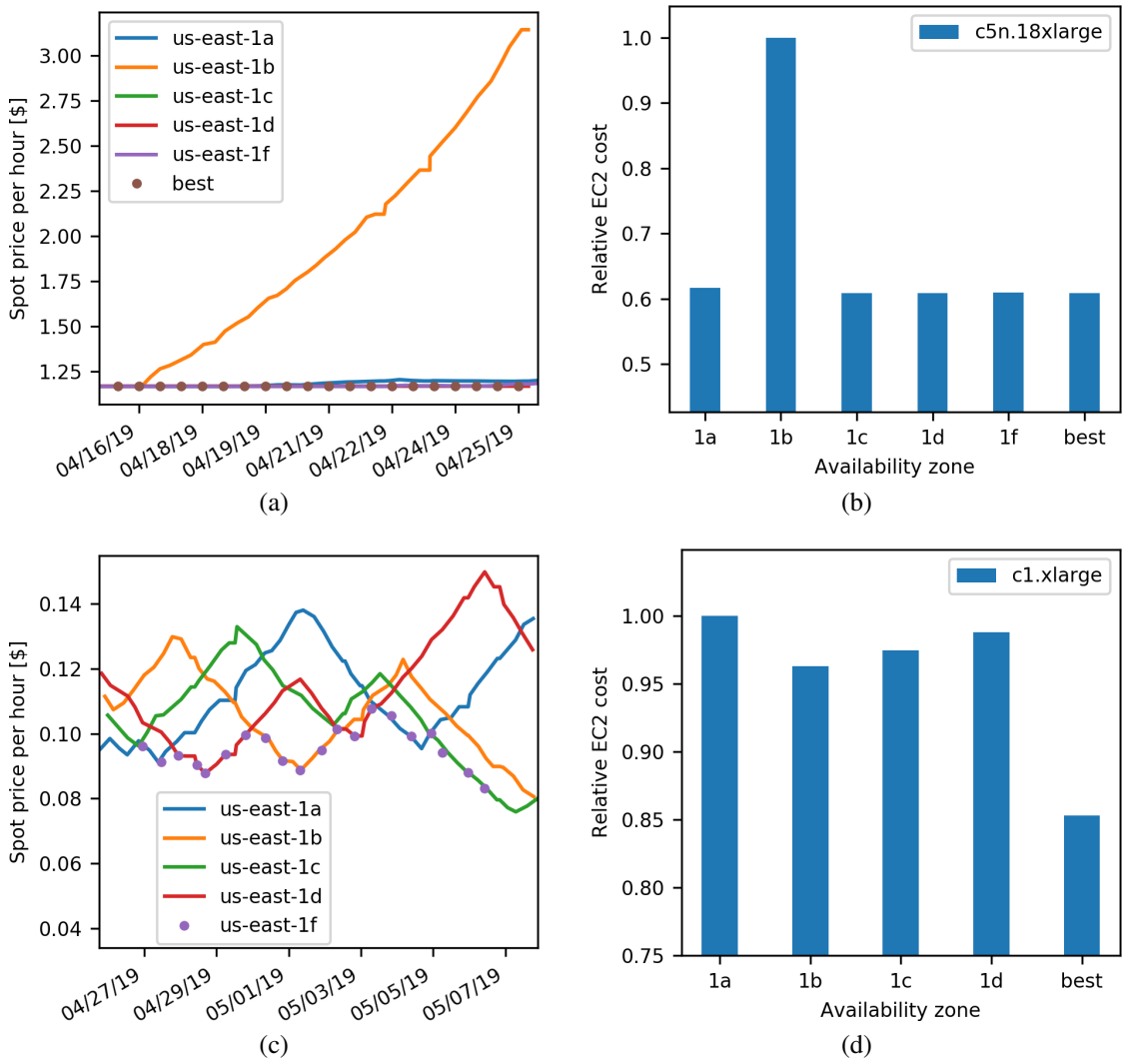


Figure 4.12: (a) Historical spot price of the `c5n.18xlarge` instance in different zones of the US East region over a 10 ten day period in April 2019. Figure (b) shows the relative cost for running an iterative seismic imaging algorithm over this time period in the respective zones. The right-most bar indicates the price for running the application with our event-driven workflow, in which the cheapest zone is automatically chosen at the start of each iteration (indicated as dots in Figure a). Figures (c) and (d) are the same plots for the `c1.xlarge` instance during a different time window.

that time period. This effects is usually negligible for programs that run in a matter of hours, as spot prices typically do not vary substantially over short periods of time. However, large-scale 3D seismic imaging problems potentially run over the course of multiple days, in which case varying spot prices can have significant influence on the cost.

We consider a hypothetical example in which we assume that we run a large-scale

imaging example for 20 iterations of an optimization algorithm, where each iteration takes 12 hours to compute, which leads to a total runtime of 10 days. Figures 4.12a shows the historical spot price of the `c5n.18xlarge` instance over a 10 day time period in April 2019 and Figure 4.12b shows the normalized cost of running the example in each specific zone. We note that the spot price of this instance type is the same across all zones at the start of the program, but that the spot price in the `us-east-1b` zone starts increasing significantly after a few hours. In this case, running the example on a fixed cluster of instances results in cost differences of 40 percent, depending on which zone was chosen at the start of the program. Our event-driven workflow with AWS Batch, allows that the cheapest available zone is automatically chosen at the start of every iteration, which ensures that zones that exhibit a sudden increase of the spot price, such as zone `us-east-1b`, are avoided in the subsequent iteration.

Another example for a different time interval and the `c1.xlarge` instance type is shown in Figure 4.12c and the relative cost of running the example in the respective zone is plotted in Figure 4.12d. The right-most bar shows the cost if the example was run with AWS Batch and the event-driven workflow had chosen the cheapest available zone at each iteration. In this case, switching zones between iterations leads to cost savings of 15 percent in comparison to running the example in the `us-east-1a` zone, which is the cheapest zone at the start of the program. For our cost estimates (Figure 4.12b and 4.12d), we assume that the spot price is not affected by our own application, i.e. by our own request of a potentially large number of spot instances, but in practice this issue needs to be taken into account as well. Overall, the two examples shown here are obviously extreme cases of price variations across zones and in practice the spot price is oftentimes reasonably stable. However, spot price fluctuations are nevertheless unpredictable and the event-driven approach with AWS Batch allows to minimize exposure to price variations.

Apart from choosing spot instances in different zones, it is also possible to vary the instance type in that is used for the computations. For example, in Figure 4.13a, we plot

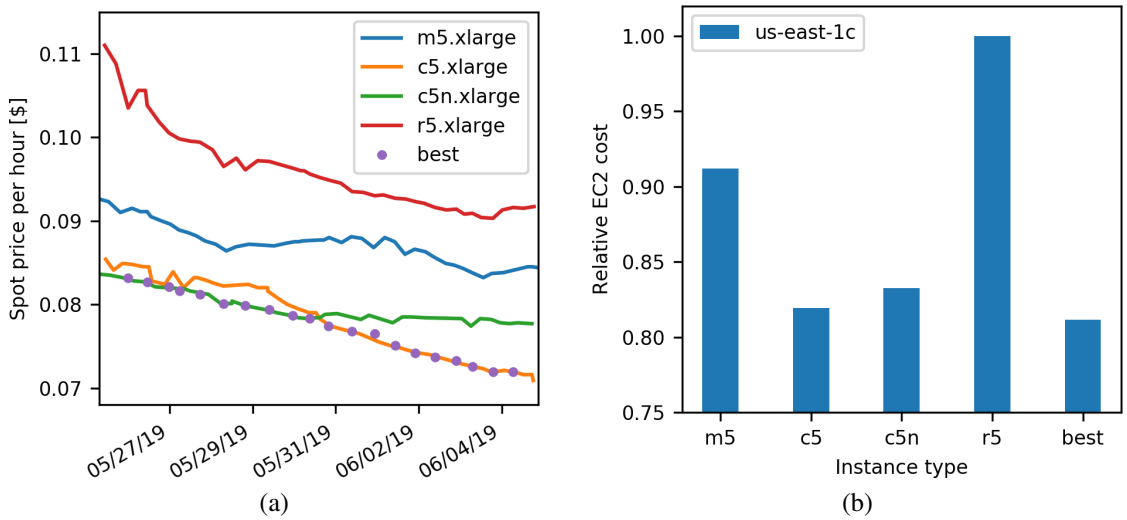


Figure 4.13: (a) Historical spot prices for a variety of `4xlarge` instances over a 10 day period in April 2019. All shown instances have 128 GB of memory, but vary in their number of CPU cores and architectures. Figure (b) shows the relative cost of running an iterative seismic imaging application over this time period in the respective zone and for the case, in which the cheapest available instance is chosen at the beginning of each iteration.

the historical spot price of the `xlarge` instance for various instance types (`m5`, `c5`, `c5n` and `r5`). All instances have 4 virtual CPUs, but vary in the amount of memory and their respective CPU architecture. Spot instances are not priced proportionally to their hardware (memory, cores, architecture), but based on the current demand. Therefore, it is oftentimes beneficial to compare different instance types and choose the currently cheapest type from a pool of possible instances. As before, we compare the relative cost for running the 10 day example on a cluster of EC2 instances, in which case the instance type is fixed for the duration of the program, against the dynamic approach with AWS Batch. Again, the event-driven approach allows to minimize exposure to price changes over the duration of the example, by choosing the cheapest available instance type at the beginning of each iteration.

4.4.5 Resilience

In the final experiment of our performance analysis, we analyze the resilience of our workflow and draw a comparison to running an MPI program on a conventional cluster of EC2 instances. Resilience is an important factor in high performance computing, especially for applications like seismic imaging, whose runtime can range from several hours to multiple days. In the cloud, the mean-time-between failures is typically much shorter than on comparable HPC systems [9], making resilience potentially a prohibiting factor. Furthermore, using spot instances further increases the exposure to instance shut downs, as spot instances can be terminated at any point in time with a two minute warning.

Seismic imaging codes that run on conventional HPC clusters typically use MPI to parallelize the sum of the source indices. MPI based applications exhibit a well known shortcoming of having a relatively low fault tolerance, as hardware failures lead to the termination of a running program. Currently, the only noteworthy approach of fault tolerance for MPI programs is the User Level Fault Mitigation (ULFM) Standard [78]. ULFM enables an MPI program to continue running after a node/instance failure, using the remaining available resources to finish the program execution. Using AWS Batch to compute the gradients $\mathbf{g}_i$ for a given batch size, provides a natural fault tolerance, as each gradient is computed by a separate container, so the crash of one instance does not affect the execution of the code on the remaining workers. Furthermore, AWS Batch provides the possibility to automatically restart EC2 instances that have crashed. In contrast to ULFM, this allows the completion of programs with the initial number of nodes or EC2 instances, rather than with a reduced number.

We illustrate the effect of instance restarts by means of our previous example with the BP model (Figure 4.5). Once again, we compute the gradient of the LS-RTM objective function for a batch size of 100 and record the runtimes without any instance/node failures. We compute the gradients using two different computational strategies for backpropagation. In the first approach, we compute the gradient on instances with a sufficient amount

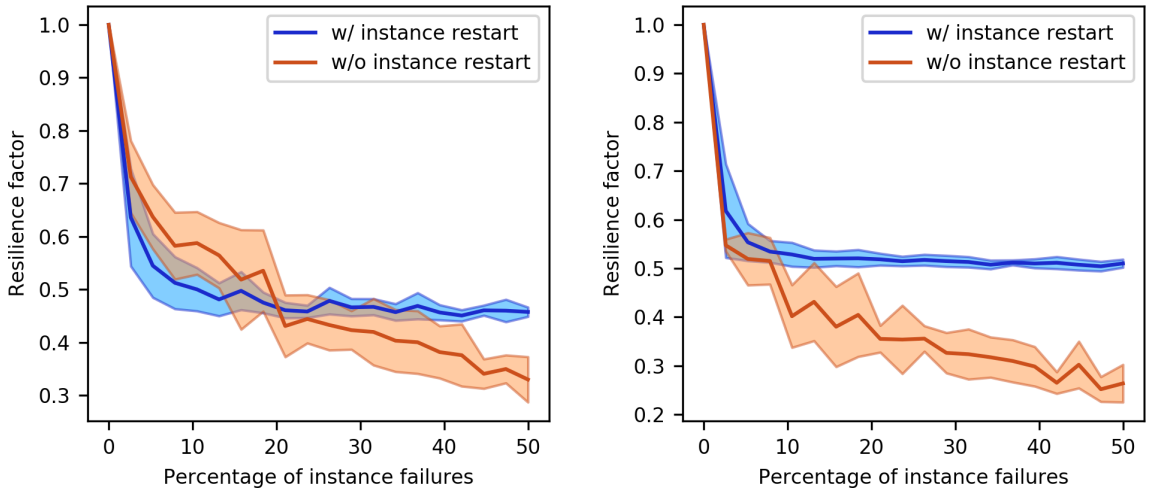


Figure 4.14: Comparison of the resilience factor (RF) for an increasing percentage of instance failures with and without instance restarts. The RF provides the ratio between the original runtime for computing the gradient of the BP model for a batch size of 100 and the runtime in the presence of instance failures. Figure (a) is the RF for an application that runs for 5 minutes without failures, while figure (b) is based on an example whose original time-to-solution is 45 minutes.

of memory to store the state variables in memory, which leads to an average runtime per gradient of 5 minutes (using 62 GB of memory). In the second approach, we compute the gradients on instances with less memory and use optimal checkpointing [42], in which case we store only a small subset of state variables and recompute the remaining states during backpropagation. This increases the average runtime per gradient to 45 minutes, but also reduces the required amount of memory for this example from 62 GB to 5 GB.

We then model the time that it takes to compute the 100 gradients for an increasing number of instance failures with and without restarts. We assume that the gradients are computed fully in parallel, i.e. on 100 parallel instances and invoke an increasing number of instance failures at randomly chosen times during the execution of program. Without instance restarts, we assign the workload of the failed instances to the workers of the remaining instances and model how long it takes complete the computation of the 100 gradients. With restarts, we add a two minute penalty to the failed workers and then restart the computation on the same instance. The two minute penalty represents the average amount of

time it takes AWS to restart a terminated EC2 instance and was determined experimentally by manually shutting down EC2 instances of an AWS Batch job and recording the time it takes to restart the container on a new instance.

Figure 4.14 shows the ratio of the time-to-solution for computing the 100 gradients without events (i.e. without failures) to the modeled time-to-solution with events. This ratio is known as the resilience factor [79] and provides a metric of how instance failures affect the time-to-solution and therefore the cost of running a given application in the cloud:

$$r = \frac{\text{time-to-solution}_{\text{ event-free}}}{\text{time-to-solution}_{\text{ event}}} \tag{4.4}$$

Ideally, we aim for this factor being as close to 1 as possible, meaning that instance failures do not significantly increase the time-to-solution. Figures 4.14a and 4.14b show the resilience factors with and without restarts for the two different backpropagation strategies, which represent programs of different runtimes. The resilience factor is plotted as a function of the percentage of instance failures and is the average of 10 realizations, with the standard deviation being depicted by the shaded colors. The plots show that the largest benefit from being able to restart instances with AWS Batch is achieved for long running applications (Figure 4.14b). The resilience factor with instance restarts approaches a value of 0.5, since in the worst case, the time-to-solution is doubled if an instance fails shortly before completing its gradient computation. Without being able to restart instances, as would be the case for MPI programs with ULFM, the gradient computations need to be completed by the remaining workers, so the resilience factor continuously decreases as the failure percentage increases. For short running applications (Figure 4.14a), the overhead of restarting instances diminishes the advantage of instance restarts, unless a significant percentage of instances fail, which, however, is unlikely for programs that run in a matter of minutes. On the other hand, long running programs or applications with a large number of workers are much more likely to encounter instance shut downs and our experiment shows that these programs benefit from the automatic instance restarts of AWS Batch.

4.5 A large-scale case study on Microsoft Azure

To complement the quantitative performance analysis of the previous section, we conclude our numerical examples with a large-scale seismic imaging case study in 3D. In contrast to our previous imaging examples based on the 2D BP model, 3D imaging involves data that is collected along a two-dimensional source and receiver grid and wave propagation is modeled in a three-dimensional domain. Accordingly, the computational cost and memory imprint of 3D imaging is multiple orders of magnitude larger than of 2D imaging, making large-scale instances of this problem oftentimes prohibitively expensive in practice. Unlike our previous examples which were conducted on AWS, we perform the following 3D imaging case study on Microsoft Azure. As such, the example not only serves the purpose to demonstrate the scalability of our framework to realistic problem sizes, but also to illustrate that the proposed event-driven approach translates to other cloud platforms as well. The following section presents a brief overview how components of our AWS workflow map to Azure and we highlight some of the (overall minor) differences. The subsequent section presents the experimental set up and imaging results.

4.5.1 Serverless imaging on Azure

For the 3D imaging case study on Azure, we re-implement the core components of the AWS workflow with the corresponding Azure services. On AWS, (image) gradients for separate source locations are computed with AWS Batch, a service for executing embarrassingly parallel containerized workloads, with the possibility of inter-node communication between virtual machines of the same job. The corresponding Azure counterpart of this service is Azure Batch [80], whose functionalities are mostly equivalent to AWS Batch and with additional support of Singularity containers [81]. This allows us to perform gradient computations on Azure in the same fashion as on AWS (Figure 4.15) and to reuse the majority of the code. Instead of Lambda functions, Azure Batch jobs are submitted using

Azure Functions, while S3 is replaced by Blob Storage [82], Azure's object storage system. Similarly, the equivalent service of AWS SQS is Azure Queue Storage [83], which is used to collect object IDs of computed gradients and to invoke their summation. The docker containers for Azure Batch are overall based on the same software stack as the containers for AWS. The only necessary changes involve swapping the AWS Python software development kit (SDK) for the Azure SDK and modifying the corresponding Python functions accordingly (e.g. for writing data to buckets/blobs).

The only notable difference between AWS and Azure Batch is the management of the computing environment. On AWS, users define the type and maximum number of EC2 instances that are available to AWS Batch, but no running instances are required prior to submitting jobs to the batch queue. In contrast, users on Azure need to launch a pool of instances prior to submitting jobs to the batch queue and the batch pool has to stay active for (at least) the duration of the job. However, as Azure Batch supports automatic resizing of pools based on certain criteria such as the number of pending jobs, the differences between AWS and Azure Batch can be circumvented in practice through the right set up of the computing environment. The second notable difference on Azure to our approach on AWS is the way in which jobs are defined and submitted to the batch queue. In our AWS workflow, job templates and submissions are managed with the AWS Python SDK, while on Azure we utilize Batch Shipyard to specify pool and job descriptions [84]. Batch Shipyard is a toolbox that provides a range of templates of docker containers, including MPI support, and enables job and pool descriptions with yaml configuration files.

The second major component of our AWS workflow is the event-driven gradient summation based on SQS and Lambda functions. On Azure we replicate this functionality using Azure Queue Storage and Azure Functions, which are direct equivalents of the AWS services. As before, workers of the Azure Batch jobs write their resulting gradients to the object storage system (blob) and send the corresponding object ID to a queue, which in turn automatically involkes the gradient summation. As the Python SDKs for Azure and AWS

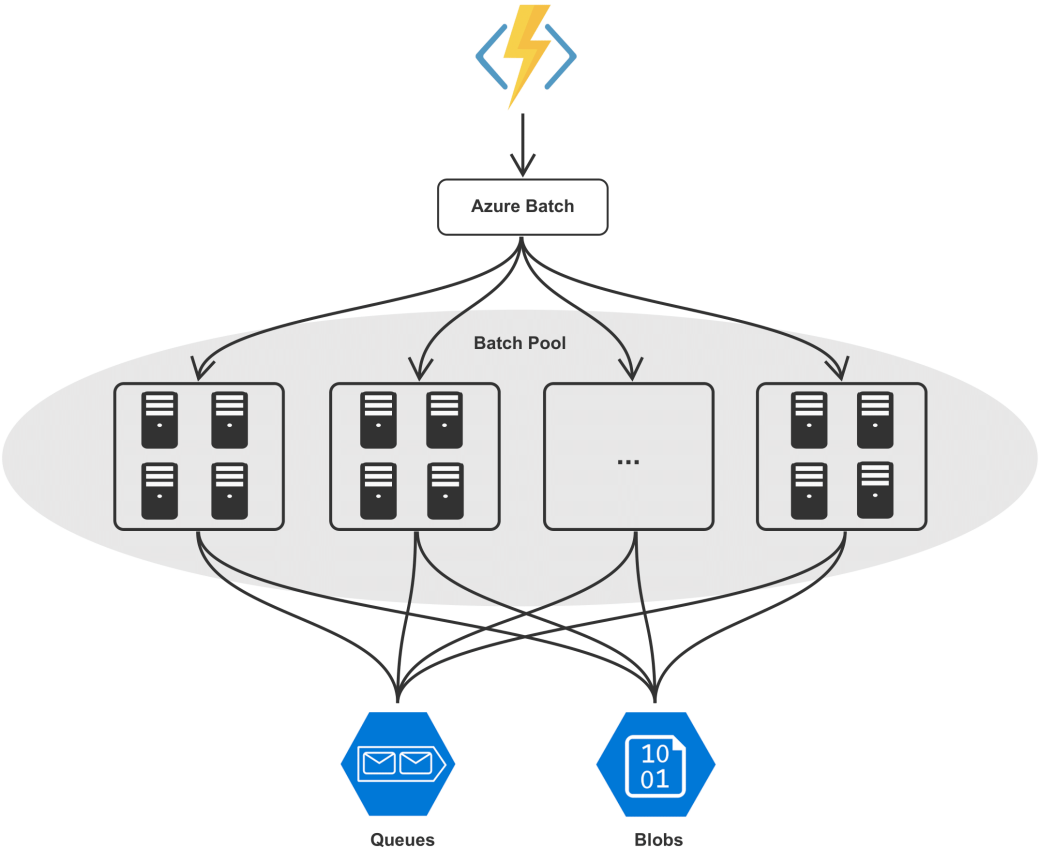


Figure 4.15: Azure setup for computing gradients of the LS-RTM objective function as a containerized embarrassingly parallel workload. As on AWS, it is possible to distribute workloads between multiple virtual machines and containers of the same job are able to communicate via message passing. Azure Blobs and Queue Storage are the equivalent services to SQS and S3 for message queuing and object storage. In contrast to AWS, Azure Batch accesses computational resources from a batch pool, which has to be launched prior to the initial submission of jobs.

are overall very similar, we can reuse our implementation of the gradient summation from AWS, which only requires swapping the S3 interface (AWS) for the blob interface (Azure). Similar to AWS, Azure Blob storage allows reading and writing objects in separate blocks, thus making it possible to process gradients that exceed the available memory of Azure Functions (1.5 to 14 GB, depending on the hosting plan) [82].

For this case study we omit the implementation of an iterative loop, as due to a budget constraint we only perform a single iteration of the imaging workflow, but using the full data set. I.e.; instead of imaging a randomly selected subset of data within an iterative stochastic gradient descent loop, we image the full data set at once, which corresponds to a single matrix-vector multiplication with the transpose of the Born scattering operator. However, in principle there are multiple possibilities to implement a serverless iterative loop on Azure. While there is no direct equivalent of AWS Step Functions, both Logic Apps [85] and Durable Functions [86] provide similar functionalities that allow orchestration of different Azure services or stateful executions of Azure Functions.

4.5.2 Reverse-time migration

In our Azure case study, we perform a single iteration of the full imaging workflow, which involves computing the gradient for every source location using Azure Batch and summing the results into a single image. This corresponds to the computation of a single full gradient, i.e. a gradient whose batchsize is equal to the total number of observed shot records. This imaging procedure is called reverse-time migration (RTM) and mathematically corresponds to applying the adjoint of the Born scattering operator to the seismic data [87]. This approach is computationally cheaper than LS-RTM as it involves a smaller number of PDE solves (namely one pass through the data), but the resulting image is blurry and typically shows an imprint of the seismic acquisition.

For our experiment, we generate a synthetic data set by modeling the observed data with the true subsurface model for a specified survey geometry. Our model and the un-

known image have dimensions of $3.325 \times 10 \times 10$ km and are discretized with a grid spacing of 12.5 m, which results in a domain of $267 \times 801 \times 801$ grid points. We generate data for a randomized seismic acquisition geometry, with data being recorded by $1,500$ receivers that are randomly distributed along the ocean floor (Figure 4.16a). The source vessel fires the seismic source on a dense regular grid as shown in Figure 4.16b, consisting of 799×799 source locations ($638,401$ in total). For seismic imaging, we assume source-receiver reciprocity, which means that sources and receivers are interchangeable and data can be sorted into $1,500$ shot records with $638,401$ receivers each. This is possible as ray paths from sources to receivers are symmetric for scalar waves and re-ordering the data has the advantage of requiring a considerably number of PDEs solves, as RTM involves two wave equations per source location. In contrast to our previous examples, which are based on the acoustic isotropic wave equation, we model wave propagation for generating and imaging the seismic data with an anisotropic acoustic wave equation [88]. The so-called *tilted transverse-isotropic* (TTI) wave equation is a more realistic representation of wave propagation in media with direction-dependent velocities, but also involves a larger memory imprint and higher computational cost due to an increased FLOP count per grid point [89, 90]. For our case study, we implement discretized versions of the forward and adjoint (linearized) TTI equation as presented in [91] using Devito.

In summary, our example involves computing seismic images for $1,500$ distinct seismic shot records using a TTI wave equation on a computational domain of $269 \times 801 \times 801$ grid points. For imaging, we use smoothed versions of the true velocity, density and anisotropy parameters, which are obtained by convolving the original models with a Gaussian kernel. The computation of every image involves solving a forward and adjoint (linearized) wave equation, and wave propagation is modeled for $3,125$ time steps. As before, computing images/gradients requires access to the forward state variables, which are thus saved in memory. To lower the memory imprint, we only save the modeled wavefield of every 10th time step in memory, which results in 313 wavefields per source location and 630 GB of

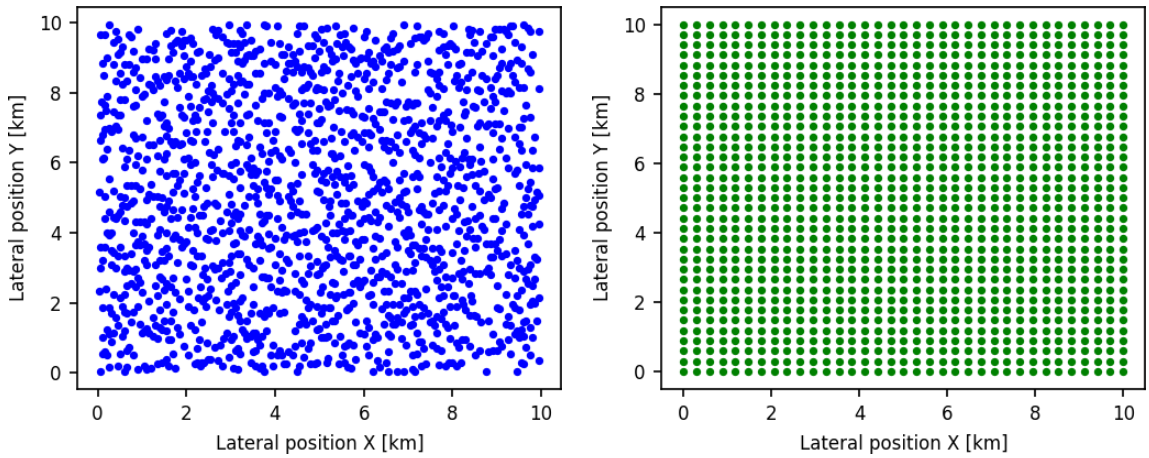


Figure 4.16: Acquisition geometry of the seismic data set that is used for the imaging case study. Figure (a) shows the receiver grid, consisting of $1,500$ randomly distributed ocean bottom sensors. Figure (b) shows the acquisition mask of the source vessel, which consists of 799×799 shot locations.

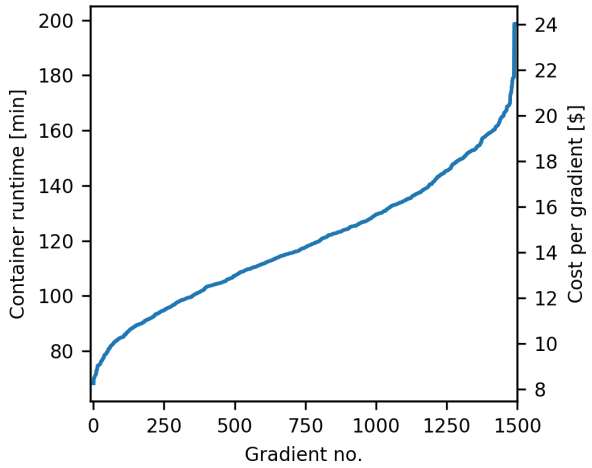


Figure 4.17: Sorted container runtimes of the Azure Batch job for each individual source location. As in the previous (2D) imaging examples, we limit the modeling domain to a 9×9 km square around the current source location, which accounts for the varying container runtimes, as sources close to the model edge are modeled on a small subset of the full model only. The right-hand axis indicates the cost for computing the individual images, each of which are computed on two E64/E64s instances.

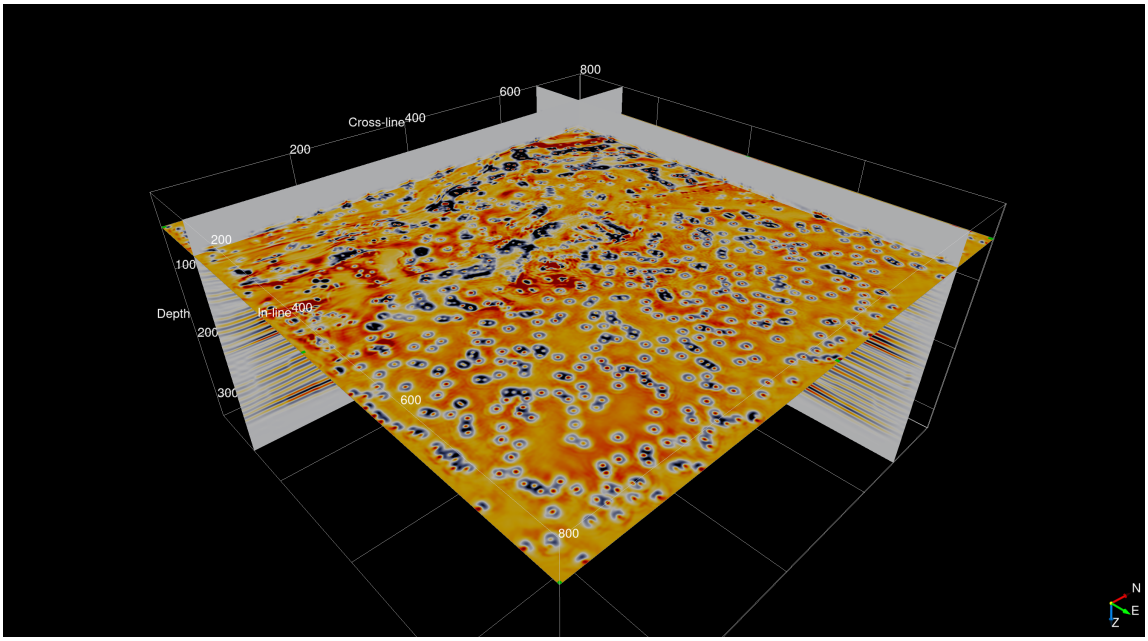
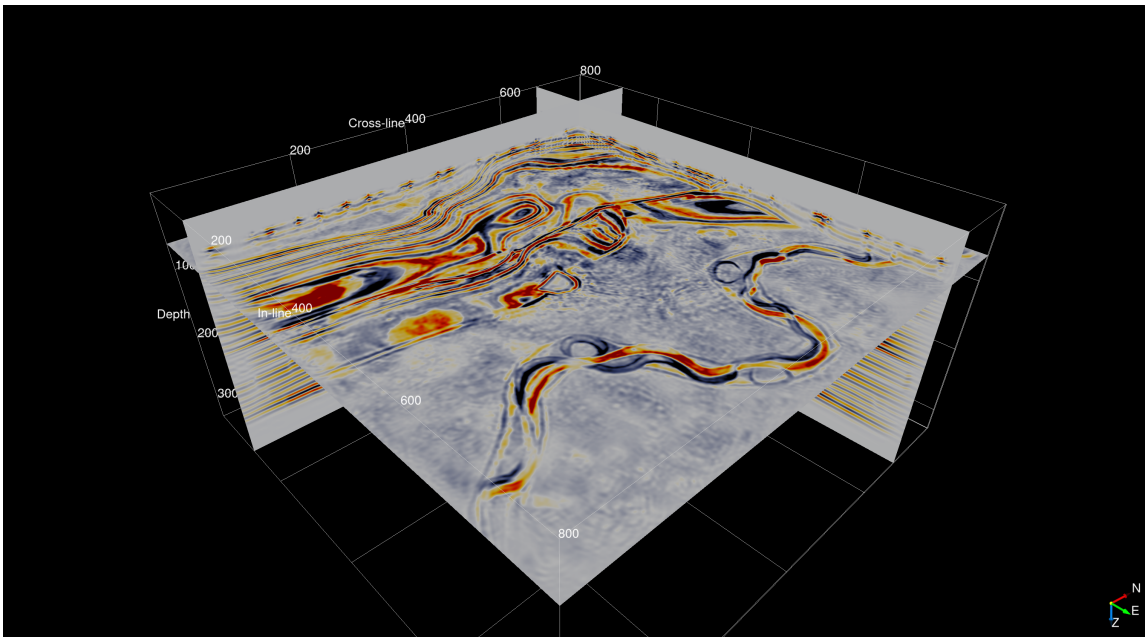
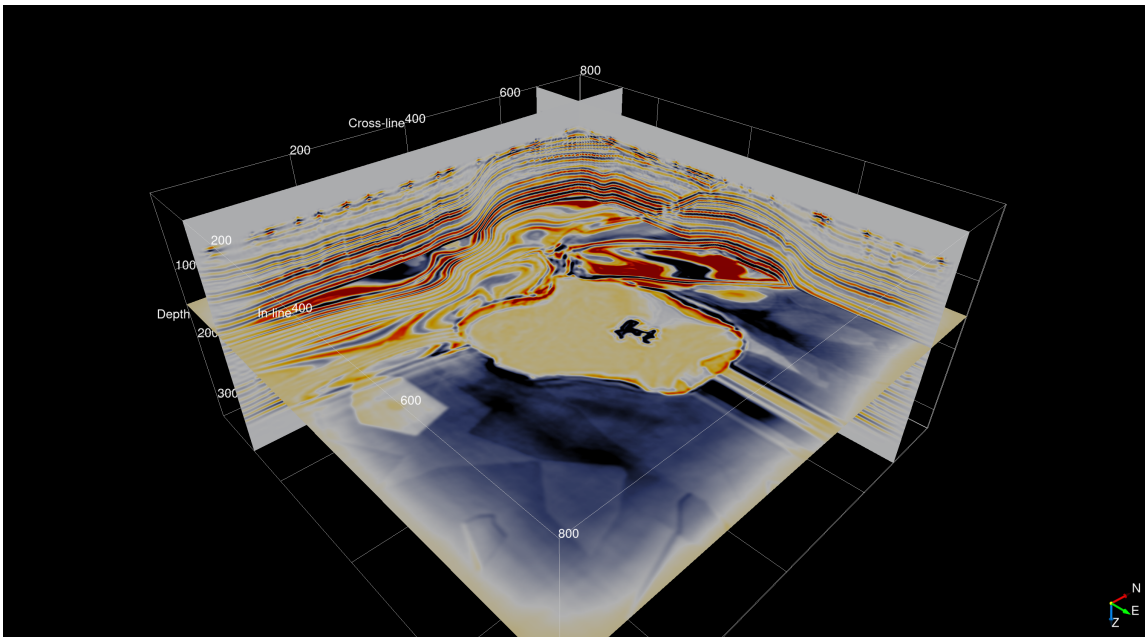


Figure 4.18: Horizontal depth slice through the final 3D image cube at 225 m depth. The shallow section of the image shows the typical source imporint, a common artifact of reverse-time migration.

required memory. For the computations, we use Azure's memory optimized E64 and E64s virtual machines, which have 432 GB of memory, 64 vCPUs and a 2.3 GHz Intel Xeon E5-2673 processor [92]. To fit the state variables in memory, we utilize two E64/E64s per source location and use MPI-based domain decomposition for solving the wave equations. As our Azure account has a quota limit of $64,000$ vCPUs, we deploy the Azure Batch job to a pool of 100 E64/E64s instances. The time-to-solution of each individual image as a function of the source location is plotted in Figure 4.17, with the average container runtime being 119.28 minutes per image. The on-demand price of the E64/E64s instances is 3.629\$ per hour, which results in a cumulative cost of $10,750$\$ for the full experiment and a total runtime of approximately 30 hours. Figures 4.18 to 4.20 show a selection of horizontal and vertical slices through the final 3D image cube, after summing the contributions from all $1,500$ source locations.

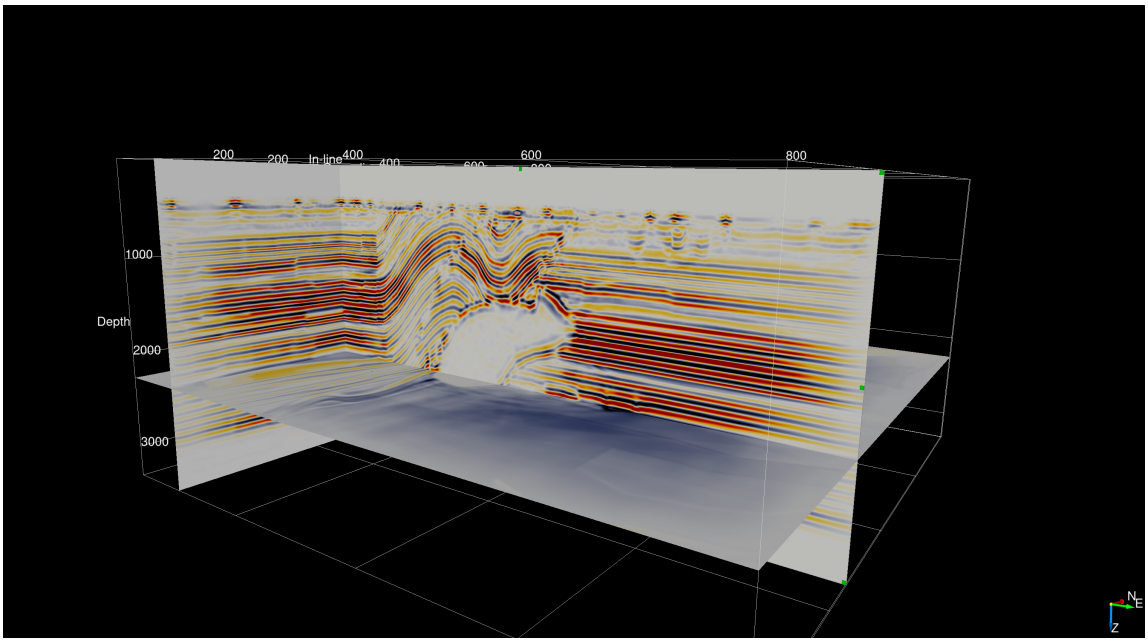


(a)

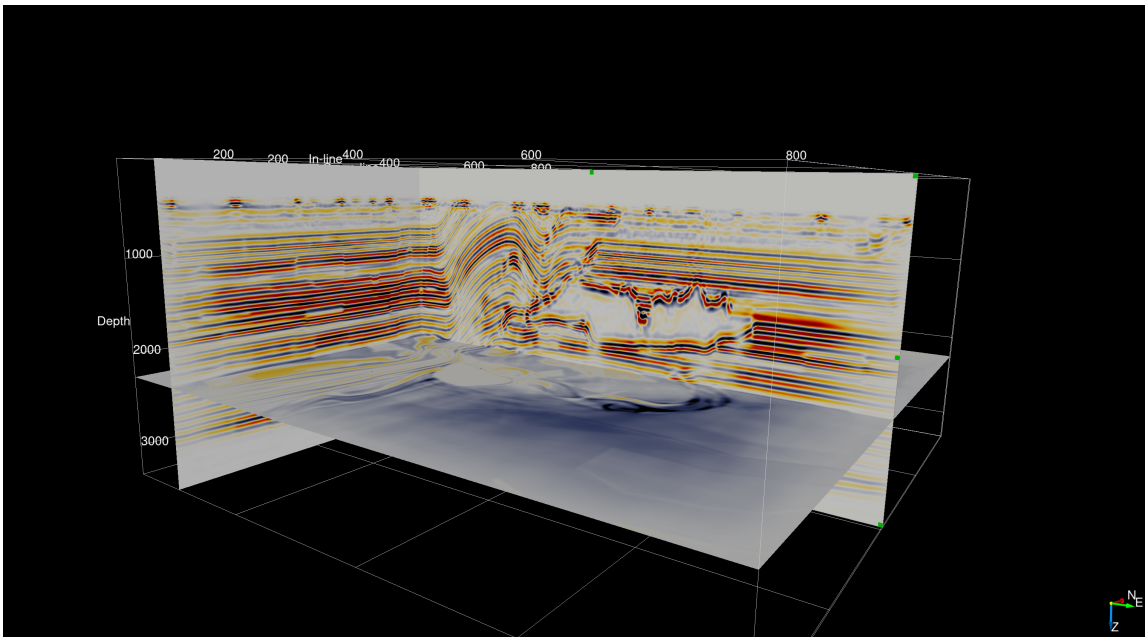


(b)

Figure 4.19: Additional horizontal slices through the final 3D image cube. Figure (a) shows a slice in the shallow part of the image at 725 m depth, at which point the source/receiver imprint is considerably weaker than at 225 m, albeit still visible. Figure (b) shows an image slice at $1,500$ m depth, at which point all acquisition artifacts are fully suppressed.



(a)



(b)

Figure 4.20: Vertical slices through the final RTM image at two distinct horizontal locations. Once again, we can observe the acquisition imprint in the shallow part of the image.

4.6 Discussion

The main advantage of an event-driven approach based on AWS Batch and Lambda functions for seismic imaging in the cloud is the automated management of computational resources by AWS. EC2 instances that are used for carrying out heavy computations, namely for solving large-scale wave equations, are started automatically in response to events, which in our case are Step Functions advancing the serverless workflow to the `ComputeGradients` state. Expensive EC2 instances are thus only active for the duration it takes to compute one element $\mathbf{g}_i$ of the full or mini-batch gradient and they are automatically terminated afterwards. Summing the gradients and updating the variables (i.e. the seismic image) is performed on cheaper Lambda functions, with billing being again solely based on the execution time of the respective code and the utilized memory. The cost overhead introduced by Step Functions, SQS messages and AWS Batch is negligible compared to the cost of the EC2 instances that are required for the gradient computations, while cost savings from spot instances and eliminating idle EC2 instances lead to significant cost savings, as shown in our examples. With the benefits of spot instances (factor $2-3$), avoidance of idle instances and the overhead of spinning clusters (factor $1.5-2$), as well as improved resilience, we estimate that our event-driven workflow provides cost savings of up to an order of magnitude in comparison to using fixed clusters of (on-demand) EC2 instances.

The second major advantage of the proposed approach is the handling of resilience. Instead of running as a single program, our workflow is broken up into its individual components and expressed through Step Function states. Parallel programs based on MPI rely on not being interrupted by hardware failures during the entire runtimetime of the code, making this approach susceptible to resilience issues. Breaking a seismic imaging workflow into its individual components, with each component running as an individual (sub-) program and AWS managing their interactions, makes the event-driven approach inherently more resilient. On the one hand, the runtime of individual components, such as computing

a gradient or summing two arrrays, is much shorter than the runtime of the full program, which minimizes the exposure to instance failures. On the other hand, AWS provides built-in resilience for all services used in the workflow. Exceptions in AWS Batch or Lambda lead to computations being restarted, while Step Functions allow users to explicitly define their behavior in case of exceptions, including the re-execution of states. Similarly, messages in an SQS queue have guaranteed at-least-once delivery, thus preventing messages from being lost. Finally, computing an embarrassingly parallel workload with AWS Batch, rather than as a MPI-program, provides an additional layer of resilience, as AWS Batch processes each item from its queue separately on an individual EC2 instance and Docker container. Instance failures therefore only affect the respective gradient and computations are automatically restarted by AWS Batch.

The most prominent disadvantage of the event-driven workflow is that EC2 instances have to be restarted by AWS Batch in every iteration of the workflow. In our performance analysis, we found that the overhead of requesting EC2 instances and starting the Docker container lies in the range of several minutes and depends on the overall batch size and on how many instances are requested per gradient. The more items are submitted to a batch queue, the longer it typically takes AWS Batch to launch the final number of instances and to process all items from the queue in parallel. On the other hand, items that remain momentarily in the batch queue, do not incur any cost until the respective EC2 instance is launched. The overhead introduced by AWS Batch therefore only increases the time-to-solution, but does not affect the cost negatively. Due to the overhead of starting EC2 instances for individual computations, our proposed workflow is therefore applicable if the respective computations (e.g. computing gradients) are both embarrassingly parallel and take a long time to compute; ideally in the range of hours rather than minutes. We therefore expect that the advantages of our workflow will be even more prominent when applied to 3D seismic data sets, where computations are orders of magnitude more expensive than in 2D. Devito provides a large amount of flexibility regarding data and model parallelism and

allows us to address the large memory imprint of backpropagation through techniques like optimal checkpointing or on-the-fly Fourier transforms, thereby presenting all necessary ingredients to apply our workflow to large-scale 3D models.

Our application, as expressed through AWS Step Functions, represents the structure of a generic gradient-based optimization algorithm and is therefore applicable to problems other than seismic imaging and full-waveform inversion. The design of our workflow lends itself to problems that exhibit a certain MapReduce structure, namely they consists of a computationally expensive, but embarrassingly parallel Map part, as well as a computationally cheaper to compute Reduce part. On the other hand, applications that rely on dense communications between workers or where the quantities of interest such as gradients or functions values are cheap to compute, are less suitable for this specific design. For example, deep convolutional neural networks (CNNs) exhibit mathematically a very similar problem structure to seismic inverse problems, but forward and backward evaluations of CNNs are typically much faster than solving forward and adjoint wave equations, even if we consider very deep networks like ResNet [93]. Implementing training algorithms for CNNs as an event-driven workflow as presented here, is therefore excessive for the problem sizes that are currently encountered in deep learning, but might be justified in the future if the dimensionality of neural networks continues to grow.

The event-driven workflow presented in this work was specifically designed for AWS and takes advantage of specialized services for batch computing or event-driven computations that are available on this platform. However, as demonstrated in the Azure case study, porting our workflow to other cloud platforms is possible as well, as almost all of the utilized services have equivalent versions on Microsoft Azure or the Google Cloud Platform (Table 4.3) [94, 95]. Services for running parallel containerized workloads in the cloud, as well as event-driven cloud functions, which are the two main components of our workflow, are available on all platforms considered in our comparison. Furthermore, both Microsoft Azure as well as the GCP offer similar Python APIs as AWS for interfacing cloud services.

Table 4.3: An overview how the AWS services used in our workflow map to other cloud providers.

Amazon Web Services	Microsoft Azure	Google Cloud
Elastic Compute Cloud	Virtual Machines	Compute Engine
Simple Storage System	Blob storage	Cloud Storage
AWS Batch	Azure Batch	Pipelines
Lambda Functions	Azure Functions	Cloud Functions
Step Functions	Logic Apps	N/A
Simple Message Queue	Queue Storage	Cloud Pub/Sub
Elastic File System	Azure Files	Cloud Filestore

We also speculate that, as cloud technology matures, services between different providers will likely grow more similar to each other. This is based on the presumption that less advanced cloud platforms will imitate services offered by major cloud providers in order to be competitive in the growing cloud market.

Overall, our workflow and performance evaluation demonstrate that cost-competitive HPC in the cloud is possible, but requires a fundamental software re-design of the corresponding application. In our case, the implementation of an event-driven seismic imaging workflow was possible, as we leverage Devito for expressing and solving the underlying wave equations, which accounts for the major workload of seismic imaging. With Devito, we are able to abstract the otherwise complex implementation and performance optimization of wave equation solvers and take advantage of recent advances in automatic code generation. As Devito generates code for solving single PDEs, with the possibility of using MPI-based domain decomposition, we are not only able to leverage AWS Batch for the parallelization over source experiments, but can also take advantage of AWS Batch's multi-node functionality to shift from data to model parallelism. In contrast, many seismic imaging codes are software monoliths, in which PDE solvers are enmeshed with IO routines, parallelization and manual performance optimization. Adapting codes of this form to the cloud is fundamentally more challenging, as it is not easily possible to isolate individual

components such as a PDE solver for a single source location, while replacing the parallelization with cloud services. This illustrates that separation of concerns and abstract user interfaces are a prerequisite for porting HPC codes to the cloud such that the codes are able to take advantage of new technologies like object storage and event-driven computations. With a domain-specific language compiler, automatic code generation, high-throughput batch computing and serverless visual algorithm definitions, our workflow represents a true vertical integration of modern programming paradimgs into a framework for HPC in the cloud.

4.7 Conclusion

Porting HPC applications to the cloud using a lift and shift approach based on virtual clusters that emulate on-premise HPC clusters, is problematic as the cloud cannot offer the same performance and reliability as conventional clusters. Applications such as seismic imaging that are computationally expensive and run for a long time, are faced with practical challenges such as cost and resilience issues, which prohibit the cloud from being widely adapted for HPC tasks. However, the cloud offers a range of new technologies such as object storage or event-driven computations, that allow to address computational challenges in HPC in novel ways. In this work, we demonstrate how to adapt these technologies to implement a workflow for seismic imaging in the cloud that does not rely on a conventional cluster, but is instead based on serverless and event-driven computations. These tools are not only necessary to make HPC in the cloud financially viable, but also to improve the resilience of workflows. The code of our application is fully redesigned and uses a variety of AWS services as building blocks for the new workflow, thus taking advantage of AWS being responsible for resilience, job scheduling, and resource allocations. Our performance analysis shows that the resulting workflow exhibits competitive performance and scalability, but most importantly minimizes idle time on EC2 instances and cost and is inherently resilient. Our example therefore demonstrates that successfully porting HPC ap-

plications to the cloud is possible, but requires to carefully adapt the corresponding codes to to the new environment. This process is heavily dependent on the specific application and involves identifying properties of the underlying scientific problem that can be exploited by new technologies available in the cloud. Most importantly, it requires that codes are modular and designed based on the principle of separation of concerns, thus making this transition possible.

REFERENCES

[1] A. A. Valenciano, "Imaging by wave-equation inversion," PhD thesis, Stanford University, 2008.

[2] S. Dong, J. Cai, M. Guo, S. Suh, Z. Zhang, B. Wang, and Z. Li, "Least-squares reverse time migration: Towards true amplitude imaging and improving the resolution," in *82nd Annual International Meeting, SEG, Expanded Abstracts*, 2012, pp. 1–5.

[3] C. Zeng, S. Dong, and B. Wang, "Least-squares reverse time migration: Inversion-based imaging toward true reflectivity," *The Leading Edge*, vol. 33, pp. 962–964, 966, 968, 9 2014.

[4] P. A. Witte, M. Louboutin, F. Luporini, G. J. Gorman, and F. J. Herrmann, "Compressive least-squares migration with on-the-fly Fourier transforms," *Geophysics*, vol. 84, no. 5, R655–R672, 2019.

[5] *Seismic processing and imaging*, `https://www.pgs.com/imaging/services/processing-and-imaging/`, 2019. (visited on 05/22/2019).

[6] *Exxonmobil sets record in high-performance oil and gas reservoir computing*, `https://corporate.exxonmobil.com/en/Energy-and-environment/Tools-and-processes/Exploration-technology/ExxonMobil-sets-record-in-high-performance-oil-and-gas-reservoir-computing`, 2019. (visited on 05/22/2019).

[7] *AWS enterprise customer success stories*, `https://aws.amazon.com/solutions/case-studies/enterprise`, 2019. (visited on 03/19/2019).

[8] A. Cockroft, "Netflix in the cloud," in *QCon San Fransisco*, Netflix, 2011.

[9] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the Amazon Web Services cloud," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 159–168.

[10] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[11] S. L. Garfinkel, "An evaluation of Amazon's grid computing services: EC2, S3, and SQS," in *Harvard Computer Science Group Technical Report TR-08-07*, 2007.

[12] J. Napper and P. Bientinesi, "Can cloud computing reach the Top500?" In *Proceedings of the Combined Workshops on Unconventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW '09)*, ser. UCHPC-MAW '09, Ischia, Italy: ACM, 2009, pp. 17–20, ISBN: 978-1-60558-557-4.

[13] K. R. Jackson, K. Muriki, L. Ramakrishnan, K. J. Runge, and R. C. Thomas, "Performance and cost analysis of the supernova factory on the Amazon AWS cloud," *Scientific Programming*, vol. 19, no. 2-3, pp. 107–119, 2011.

[14] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu, "Magellan: Experiences from a science cloud," in *Proceedings of the 2nd International Workshop on Scientific Cloud Computing*, ACM, 2011, pp. 49–58.

[15] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright, "Evaluating interconnect and virtualization performance for high performance computing," *Association for Computing Machinery (ACM) SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 55–60, 2012.

[16] S. Benedict, "Performance issues and performance analysis tools for HPC cloud applications: A survey," *Computing*, vol. 95, no. 2, pp. 89–108, 2013.

[17] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas, "Performance evaluation of Amazon Elastic Compute Cloud for NASA high-performance computing applications," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 4, pp. 1041–1055, 2016.

[18] A. Gupta and D. Milojicic, "Evaluation of HPC applications on cloud," in *2011 Sixth Open Cirrus Summit*, IEEE, 2011, pp. 22–26.

[19] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, Apr. 2017.

[20] C. Kotas, T. Naughton, and N. Imam, "A comparison of Amazon Web Services and Microsoft Azure cloud platforms for high performance computing," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, IEEE, 2018, pp. 1–4.

[21] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[22] P. Rad, A. Chronopoulos, P. Lama, P. Madduri, and C. Loader, "Benchmarking bare metal cloud servers for HPC applications," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, IEEE, 2015, pp. 153–159.

[23] M. Mohammadi and T. Bazhirov, "Comparative benchmarking of cloud computing vendors with high performance LINPACK," in *Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications (HP3C-2018)*, ACM, 2018, pp. 1–5.

[24] W. Gropp, W. D. Gropp, A. D. F. E. E. Lusk, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1999, vol. 1.

[25] *AWS Documentation: AWS Lambda*, `https://aws.amazon.com/lambda/`, 2019. (visited on 06/01/2019).

[26] *AWS Documentation: Amazon Simple Storage Service*, `https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html`, 2019. (visited on 07/25/2019).

[27] *Google Cloud Storage*, `https://cloud.google.com/storage/`, 2019. (visited on 07/25/2019).

[28] M. Barton, W. Reese, J. A. Dickinson, J. B. Payne, C. B. Thier, and G. Holt, *Massively scalable object storage system*, US Patent 9,021,137 to Rackspace US, Inc, Apr. 2015.

[29] A. Friedman and A. Pizarro, *Building high-throughput genomics batch workflows on AWS*, `https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-introduction-part-1-of-4/`, May 2017.

[30] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the Association for Computing Machinery (ACM)*, vol. 51, no. 1, pp. 107–113, 2008.

[31] *AWS Documentation: AWS Batch*, `https://aws.amazon.com/ec2/`, 2019. (visited on 03/19/2019).

[32] *Apache Hadoop*, `https://hadoop.apache.org/`, 2019. (visited on 08/12/2019).

[33] *AWS Documentation: Amazon EMR*, https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-overview.html, 2019. (visited on 08/12/2019).

[34] F. Billette and S. Brandsberg-Dahl, "The 2004 BP velocity benchmark.," in *67th Annual International Meeting, EAGE, Expanded Abstracts*, EAGE, 2005, B035.

[35] *Advances in seismic imaging technology*, https://www.hartenergy.com/exclusives/advances-seismic-imaging-technology-177370, 2019. (visited on 06/04/2019).

[36] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, no. 8, p. 1259, 1984.

[37] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, WCC127–WCC152, Nov. 2009.

[38] R. G. Pratt, "Seismic waveform inversion in the frequency domain, Part 1: Theory and verification in a physical scale model," *Geophysics*, vol. 64, no. 3, pp. 888–901, 1999.

[39] B. Peters, B. R. Smithyman, and F. J. Herrmann, "Projection methods and applications for seismic nonlinear inverse problems with multiple constraints," *Geophysics*, vol. 84, no. 2, R251–R269, 2019.

[40] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *International Business Machines Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, Mar. 1967.

[41] L. Ruthotto and E. Haber, *Deep neural networks motivated by partial differential equations*, http://arxiv.org/abs/1804.04272, Computing Research Repository (arXiv CoRR), 2018. (visited on 09/12/2018).

[42] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *Association for Computing Machinery (ACM) Transactions on Mathematical Software*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[43] C. M. Furse, "Faster than Fourier-ultra-efficient time-to-frequency domain conversions for FDTD," in *Institute of Electrical and Electronics Engineers (IEEE) Antennas and Propagation Society International Symposium*, vol. 1, Jun. 1998, 536–539 vol.1.

[44] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *2009 International Conference on High Performance Computing Simulation*, Jun. 2009, pp. 36–43.

[45] R. M. Weiss and J. Shragge, "Solving 3D anisotropic elastic wave equations on parallel GPU devices," *Geophysics*, vol. 78, no. 2, F7–F15, 2013.

[46] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[47] W. W. Symes, D. Sun, and M. Enriquez, "From modelling to inversion: Designing a well-adapted simulator," *Geophysical Prospecting*, vol. 59, no. 5, pp. 814–833, 2011.

[48] L. Ruthotto, E. Treister, and E. Haber, "jInv–a flexible Julia package for PDE parameter estimation," *Society for Industrial and Applied Mathematics (SIAM) Journal on Scientific Computing*, vol. 39, no. 5, S702–S722, 2017.

[49] C. D. Silva and F. J. Herrmann, "A unified 2D/3D large-scale software environment for nonlinear inverse problems," *Association for Computing Machinery (ACM) Transactions on Mathematical Software (TOMS)*, vol. 45, 7:1–7:35, 2017.

[50] P. A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G. J. Gorman, and F. J. Herrmann, "A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia," *Geophysics*, vol. 84, no. 3, F57–F71, 2019.

[51] *AWS Documentation: How spot instances work*, `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/how-spot-instances-work.html`, 2019. (visited on 03/21/2019).

[52] Y. Nesterov, *Lectures on convex optimization*. Springer, 2018, vol. 137.

[53] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, `https://arxiv.org/abs/1412.6980`, arXiv e-prints, 2014. (visited on 05/07/2017).

[54] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 1964.

[55] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *Society for Industrial and Applied Mathematics (SIAM) Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[56] *AWS Documentation: Amazon Elastic Compute Cloud*, https://docs.aws.amazon.com/batch/latest/userguide/what-is-batch.html, 2019. (visited on 07/26/2019).

[57] *Starcluster*, http://star.mit.edu/cluster/, 2019. (visited on 06/05/2019).

[58] *AWS High Performance Computing*, https://aws.amazon.com/hpc/, 2019. (visited on 06/05/2019).

[59] *AWS Documentation: AWS Step Functions*, https://aws.amazon.com/step-functions/, 2019. (visited on 07/26/2019).

[60] *Iterating a loop using Lambda*, https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html, 2018. (visited on 06/12/2018).

[61] *Docker*, https://www.docker.com/, 2019. (visited on 07/26/2019).

[62] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.

[63] *AWS Documentation: Amazon Simple Queue Service*, https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html, 2019. (visited on 07/25/2019).

[64] *AWS Documentation: Amazon EC2 instance types*, https://aws.amazon.com/ec2/instance-types/, 2019. (visited on 07/30/2019).

[65] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. A. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, *Architecture and performance of Devito, a system for automated stencil computation*, https://arxiv.org/abs/1807.03032, Computing Research Repository (arXiv CoRR), 2018. (visited on 07/21/2018).

[66] D. Joyner, O. Certik, A. Meurer, and B. E. Granger, "Open source computer algebra systems: SymPy," *Association for Computing Machinery (ACM) Communications in Computer Algebra*, vol. 45, no. 3/4, pp. 225–234, Jan. 2012.

[67] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, *High-level Python abstractions for optimal checkpointing in inversion problems*, https://arxiv.org/abs/1802.02474, Computing Research Repository (arXiv CoRR), Jan. 2018. (visited on 02/14/2018).

[68] P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage, and F. J. Herrmann, "Event-driven workflows for large-scale seismic imaging in the cloud," in *89th Annual International Meeting, SEG, Expanded Abstracts*. 2019, pp. 3984–3988.

[69] J. Rad, A. Ragab, and A. Damodar, *Building a tightly coupled molecular dynamics workflow with multi-node parallel jobs in AWS Batch*, https://aws.amazon.com/blogs/compute/building-a-tightly-coupled-molecular-dynamics-workflow-with-multi-node-parallel-jobs-in-aws-batch/, Nov. 2018. (visited on 10/25/2018).

[70] *Boto 3 documentation*, https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, 2019. (visited on 07/26/2019).

[71] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[72] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of Computational Statistics 2010: 19th International Conference on Computational Statistics Paris France*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[73] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using Docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, IEEE, 2016, pp. 52–57.

[74] A. Valli and A. Quarteroni, *Domain decomposition methods for partial differential equations*. Numerical Mathematics and Scientific Computation. The Clarendon Press, Oxford University Press, New York, 1999.

[75] *AWS Documentation: AWS Batch - Multi node parallel jobs*, https://docs.aws.amazon.com/batch/latest/userguide/multi-node-parallel-jobs.html, 2019. (visited on 07/29/2019).

[76] *Amazon EC2 C5 instances*, https://aws.amazon.com/ec2/instance-types/c5/, 2019. (visited on 06/12/2019).

[77] *Announcing accelerated scale-down of AWS Batch managed compute environments*, https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-accelerated-scale-down-of-aws-batch-managed-compute-environments/, 2019. (visited on 06/21/2019).

[78] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[79] S. Hukerikar, R. A. Ashraf, and C. Engelmann, "Towards new metrics for high-performance computing resilience," in *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*, ACM, 2017, pp. 23–30.

[80] *Azure Batch documentation*, https://docs.microsoft.com/en-us/azure/batch/, 2019. (visited on 11/05/2019).

[81] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *Plos One*, vol. 12, no. 5, e0177459, 2017.

[82] *Introduction to Azure Blob storage*, https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction, 2019. (visited on 11/05/2019).

[83] *Introduction to Azure Storage*, https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction, 2019. (visited on 11/05/2019).

[84] *Batch Shipyard*, https://github.com/Azure/batch-shipyard, 2019. (visited on 11/06/2019).

[85] *Overview - What is Azure Logic Apps?* https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview, 2019. (visited on 11/06/2019).

[86] *What are Durable Functions?* https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview, 2019. (visited on 11/06/2019).

[87] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, Nov. 1983.

[88] Y. Zhang, H. Zhang, and G. Zhang, "A stable TTI reverse time migration and its implementation," *Geophysics*, vol. 76, no. 3, WA3–WA11, 2011.

[89] R. Fletcher, X. Du, and P. J. Fowler, "A new pseudo-acoustic wave equation for TI media," in *78th Annual International Meeting, SEG, Expanded Abstracts*. 2008, pp. 2082–2086.

[90] C. Chu, B. K. Macy, and P. D. Anno, "An accurate and stable wave equation for pure acoustic TTI modeling," in *81st Annual International Meeting, SEG, Expanded Abstracts*. 2011, pp. 179–184.

[91] M. Louboutin, P. Witte, and F. J. Herrmann, "Effects of wrong adjoints for RTM in TTI media," in *88th Annual International Meeting, SEG, Expanded Abstracts*. 2018, pp. 331–335.

[92] *Linux virtual machines pricing*, `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/`, 2019. (visited on 11/06/2019).

[93] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on Computer Bision and Pattern Recognition*, 2016, pp. 770–778.

[94] *AWS to Azure services comparison*, `https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services`, 2019. (visited on 07/30/2019).

[95] *Google Cloud Platform for AWS professionals*, `https://cloud.google.com/docs/compare/aws/`, 2019. (visited on 07/30/2019).

CHAPTER 5
CONCLUSIONS

In this thesis, I have made three contributions that address computational and software-related challenges in the field of seismic inverse problems. Two important applications that arise in this field are nonlinear seismic parameter estimation (full-waveform inversion) and the closely related linearized inverse problem of seismic imaging (least squares reverse-time migration). These problems draw their complexity from a combination of computational challenges, namely the high cost of repeatedly solving wave equations, and algorithmic challenges resulting from their mathematical properties, such as non-convexity or ill-conditioning. This thesis addresses the question how software for seismic inverse problems needs to be designed in order to stimulate algorithmic innovation, while at the same time providing performance to work on realistically sized problems. Furthermore, I address the question how redundancy and (transform-domain) sparsity of seismic data and images can be exploited to address the high computational cost of wave-equation-based seismic imaging, using recent breakthroughs in signal processing and convex optimization. Finally, I address the question of how the cloud can be adapted for high-performance applications like seismic imaging. Even though hardware and data centers used in cloud computing closely resemble on-premise HPC clusters, there is a fundamental difference in how these features are exposed to users and accordingly, how they can be used most effectively in terms of cost, resilience and turn-around time. In the final chapter of this thesis, I summarize the findings and conclusions that were presented throughout this thesis in regard to these research questions.

5.1 Software for seismic inverse problems

Writing software for applications like seismic imaging and parameter estimation is an enormously complex task, as it requires knowledge in an intersection of multiple scientific fields ranging from physics and numerical analysis to optimization, high-performance computing and compiler technologies. In principle, this requires a close multi-disciplinary collaboration between domain experts of various fields, but in practice we find that codes for inversion or differential equations are oftentimes solely developed by researchers from a single domain. This leads to software that either does not satisfy special requirements of domains scientists or to codes with poor design and performance. Being able to manage the complexity of software for high-performance applications in CSE is therefore an ongoing challenge and active field of research.

Chapter 2 introduces a framework in the Julia programming language for seismic modeling and inversion, which manages the complexity of seismic inversion codes through abstract user interfaces and a vertical integration of compiler technologies and automatic code generation. The Julia framework exposes solvers for forward and adjoint wave equations through matrix-free linear operators, thereby enabling the implementation of algorithms that closely resemble the mathematical notation. This is combined with out-of-core abstract data containers, for which common mathematical operations (inner products, norms, etc.) are overloaded. This allows treating seismic data as Julia vectors without having to keep the full (multi-dimensional) data volume in memory. The underlying wave equations are implemented with Devito, a domain-specific language compiler in Python with an API for symbolic definitions of PDEs based on SymPy [1] expressions. For solving forward and adjoint wave equations, the Devito compiler performs a series of performance optimizations on the symbolic expressions and generates optimized C code on the fly. My Julia framework therefore represents a collaborative effort of geophysicists, mathematicians and compiler experts and a true vertical integration of modern code generation and compiler

technologies into a geophysical inversion framework. Managing the complexity of a seismic inversion code is thus managed through abstract user interfaces and a separation of concerns, which not only enables unit testing, but also facilitates modifying and extending individual components of the software. Through a series of numerical examples, I demonstrate that this allows high-level symbolic implementations of various seismic inversion algorithms for large-scale problem sizes.

5.2 Algorithms for seismic imaging

While seismic imaging is, in contrast to full waveform inversion, a linear least squares problem, its complexity results from poor conditioning, inconsistency and the even larger computational cost, as data is processed at higher frequencies and requires smaller grid intervals. Classic matrix-free solvers like the nonlinear conjugate gradient method in principle require the computation of the gradient as a sum over all seismic source positions, which is infeasible for problem sizes encountered in practice. Similar to deep learning, optimization algorithms can generally only be run for a fixed number of passes through the data (epochs) and therefore rely on stochastic sampling and randomization techniques [e.g. 2, 3]. In the scenario where optimization algorithms cannot be run to convergence, results are especially sensitive to the behavior of optimization algorithms during the early iterations and the choice of hyperparameters, such as regularization parameters. Additionally, seismic imaging requires computationally effective strategies for accessing forward state variables during backpropagation to reduce the memory imprint and avoid having to write wavefields to disk [4, 5].

Chapter 3 introduces an algorithm for seismic imaging that addresses these challenges by combining on-the-fly Fourier transforms with ideas from compressive sensing and sparsity-promoting minimization. Instead of computing gradients in the time domain using backpropagation, the approach is based on on-the-fly discrete Fourier transforms (DFTs) to compute gradients in the frequency domain, where the memory imprint only depends on

the number of frequencies for which the gradient is computed, as gradients are separable over frequencies. To limit the computational cost and required memory, the gradient is computed only for a small number of randomly selected frequencies and source locations during each iteration, which effectively leads to an underdetermined problem and noisy image updates. By formulating the problem as a sparsity-promoting minimization problem in which we exploit the fact that seismic images can be well approximated by a small number of curvelet coefficients, it is possible to remove interference noise from source and frequency subsampling as part of the inversion. Unlike conventional algorithms in the time-domain based on optimal checkpointing or boundary wavefield reconstruction, the computational cost and memory imprint of this approach only depend on the batchsize of sources and frequencies, which in turn depend on the sparsity and size of the image, but not on the number of modeled time steps.

Sparsity-promoting minimization in combination with on-the-fly DFTs and random subsampling are therefore very powerful tools for seismic imaging, as they allow us to considerably reduce computational cost and memory demand, while results are qualitatively comparable to images that are computed with conventional time-domain methods. Specifically, we find that speedups of up to a factor of four can be achieved through on-the-fly DFTs in comparison to optimal checkpointing, which involves recomputing the forward state variables from a number of checkpoints. Future research directions could involve comparisons to other alternatives, such as writing wavefields to disk using various compression libraries. Additional speedups in comparison to full-gradient methods such as the nonlinear conjugate gradient method [6] are achieved by limiting the number of data passes (epochs) to a small number, which is possible as seismic data is generally oversampled and sparsity-promoting algorithms remove inference and subsampling artifacts as part of the inversion. A major difference between optimal checkpointing and sparsity-promoting imaging with on-the-fly DFTs are the computational trade-offs that the respective approaches provide. Namely, optimal checkpointing provides a trade-off between memory and com-

pute, as more time steps have to be modeled if a smaller number of wavefield checkpoints are saved in memory (or on disk). In contrast, compressive imaging with on-the-fly DFTs trades both compute and memory for noisier image updates and therefore a larger number of iterations to arrive at an acceptable solution.

5.3 Seismic imaging in the cloud

The final chapter of this thesis proposes a new model for running large-scale HPC applications like seismic imaging in the cloud. Due to the high computational cost that is associated with repeatedly solving wave equations for seismic inverse problems, access to HPC resources is necessary to work on relevant problem sizes, but this infrastructure is often not available to research groups or small companies. The cloud generally offers appropriate hardware and infrastructure to address HPC applications such as seismic imaging, but some fundamental differences need to be taken into consideration in order to use resources cost effectively and circumvent certain shortcomings. In particular, the cloud can generally not offer the same amount of bandwidth and latency as comparable HPC clusters, or only at a considerable amount of cost, as standard cloud VMs are based on Ethernet technology as opposed to high-speed connections fabrics such as InfiniBand. Other important factors include oftentimes inferior resilience (i.e. a smaller mean time between failures) and a variety of pricing models, such as usage-based or auction-based pricing (spot instances). Due to these factors, porting legacy software designed for on-premise HPC clusters to the cloud and running it on clusters of virtual machines is not an effective approach and potentially results in deteriorating performance and high cost. Thus, the fundamental research question of chapter 4, is how the cloud can be adapted effectively for HPC applications and how we can take advantage of technological advances presented by it.

My approach presented in chapter 4 is based on a serverless architecture, in which computational resources are managed by the cloud environment and virtual machines only run exactly as long as they are utilized. By interpreting iterative algorithms for seismic imag-

ing as a series of map-reduce problems, I can take advantage of existing cloud technologies such as batch computing services and event-driven computations. Specifically, I use AWS Batch to compute the gradient of the LS-RTM objective function, which takes advantage of the fact that the computations for different source positions are embarrassingly parallel. AWS Batch processes jobs from the batch queue as individual docker containers and automatically starts and terminates the required number of EC2 instances, thereby avoiding idle resources. The summation of the gradients, i.e. the reduce part, is implemented with AWS Lambda functions and takes advantage of the fact that the duration of computing individual components of the gradient varies and that results arrive in the object storage system over a period of multiple minutes (or even hours). The seismic imaging case study on Azure shows that my event-driven map-reduce architecture also translates to other cloud platforms, making this approach not only valid on AWS.

From my performance analysis, which shows that a reasonable trade-off between cost and performance is achievable in this fashion, I conclude that adapting the cloud for HPC applications like seismic imaging is possible, but requires to re-architecture the corresponding software. This task is strongly application dependent and requires an analysis how underlying problem structures can be exploited by cloud tools or services. If this is taken into account, the cloud offers a large amount of flexibility regarding hardware, services and pricing models, which allow addressing computational challenges of HPC applications in novel ways. Once again, the prerequisite from the software side to make this kind of transition possible, is code with a hierarchically structure based on abstract user interfaces. In our case, this makes it for examples possible to exchange the Julia implementation of the parallelization over sources from chapter 2 with AWS Batch and therefore benefit from containerization and automatic resource allocations. My experiences with AWS and Azure therefore underline my earlier conclusions, namely that software based on separation of concerns is a prerequisite for managing the high complexity of HPC and CSE applications and for facilitating algorithmic innovation.

REFERENCES

[1] D. Joyner, O. Certik, A. Meurer, and B. E. Granger, "Open source computer algebra systems: SymPy," *Association for Computing Machinery (ACM) Communications in Computer Algebra*, vol. 45, no. 3/4, pp. 225–234, Jan. 2012.

[2] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of Computational Statistics 2010: 19th International Conference on Computational Statistics Paris France*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[3] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, `https://arxiv.org/abs/1412.6980`, arXiv e-prints, 2014. (visited on 05/07/2017).

[4] G. A. McMechan, "Migration by extrapolation of time-dependent boundary values," *Geophysical Prospecting*, vol. 31, no. 3, pp. 413–420, 1983.

[5] W. W. Symes, "Reverse time migration with optimal checkpointing," *Geophysics*, vol. 72, no. 5, SM213–SM221, 2007.

[6] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 1964.

CHAPTER 6

OUTLOOK AND FUTURE DIRECTIONS

6.1 Integration of JUDI into deep learning frameworks

An interesting direction of future research is the integration of seismic modeling codes into packages for deep learning to enable combined data-model-driven approaches for seismic inverse problems. Efforts in this direction have been undertaken in the context of medical imaging [1, 2] and ordinary differential equations (ODEs) [3, 4], but it remains an open question how this can be achieved in the context of seismic inversion. Some authors have pointed out that time stepping for acoustic modeling can be cast as a feed-forward convolutional neural network (CNN) and implemented in frameworks like PyTorch or Tensorflow [5, 6]. This in turn allows leveraging these framework's automatic differentiation tools to compute gradients of FWI objective functions automatically. However, deep learning frameworks are fundamentally designed and optimized for dense linear algebra operations and convolutions are conventionally computed as explicit matrix-matrix multiplications using for example the general matrix multiply (GeMM) [7] or Winograd algorithm [8]. Efficient time stepping codes for acoustic modeling on the other hand, such as the code generated by Devito, are stencil based, which is favorable in settings with large domains/ images and a small number of channels (namely single channels).

This leads to the interesting question how stencil-based codes for time stepping can be integrated into deep learning frameworks and used for both forward and backward network evaluations (i.e. backpropagation). I argue that once again, this can be achieved by exploiting representations of seismic modeling operators through high-level abstractions like matrix-free linear operators, which can be relatively easily integrated into deep learning frameworks. In the following section, I present initial steps towards integrating my

Julia framework from chapter 2 into the *Flux* deep learning library [9] and discuss possible extensions and applications.

6.1.1 Linear and nonlinear JUDI operators with Flux

My basic goal is to enable compositions of neural network layers (e.g. convolutional or fully-connected layers) with operators for seismic modeling. Instead of re-implementing seismic modeling operators with convolutions from machine learning libraries, I want to use my own custom modeling operators, namely JUDI operators for Born- and nonlinear modeling. Even more importantly, I want to evaluate my own expressions during back-propagation, but fully integrate them into Flux's automatic differentiation (AD) module (Flux.Tracker).

For linear operators such as the Born scattering operator (Jacobian), this can be achieved fairly easily as the forward pass simply corresponds to the matrix-vector product of the Jacobian `J` with the respective input vector. Accordingly, backpropagation involves computing the matrix-vector product of the adjoint Jacobian with the incoming residual vector Δ. As multiplications with `J` are already overloaded in JUDI, all that is left to do is overload the gradient macro of Flux and specify the respective operations for the forward and backward pass (Listing 6.1).

```
1 @grad J::judiJacobian * x::AbstractVecOrMat =
2         Tracker.data(J)*Tracker.data(x), Δ -> (nothing, adjoint↩
              (J) * Δ)
```

Listing 6.1: To enable backpropagation with Flux through layers with linearized Born scattering operators, the gradient macro is overloaded for the respective `judiJacobian` data type. Line 2 defines the operations for the forward and backward pass, which correspond to matrix-vector products of the forward and adjoint Jacobian with the respective input vectors.

The code in Listing 6.1 instructs Flux to compute the gradient with respect to the input vector `x` using the adjoint Born operator, which is precisely a JUDI time-stepping operator.

The derivative with respect to the Jacobian itself is intentionally left empty (`nothing`), as this avoids forming a dense matrix of the size of `J`. Having overloaded the gradient macro for the `judiJacobian` operator, it is now possible to use the Jacobian as part of a network and combine it with Flux utilities for computing misfits and gradients (Listing 6.2).

```
# Test demigration operator w/ Flux Dense Layer
y = randn(Float32, 200)
W = randn(Float32, 100, length(y))
b = randn(Float32, length(y))

# Linearized Born operator
J = judiJacobian(F, q)

# Example image
x = vec(image)

predict(x) = W*(J*x) .+ b
loss(x,y) = Flux.mse(predict(x), y)

# Compute gradient w/ Flux
gs = Tracker.gradient(() -> loss(x, y), params(W, b, x))
gs[x]   # evalute gradient of x
```

Listing 6.2: A shallow neural network consisting of the Born scattering operator `J` and a fully connected layer. By overloading the gradient for `J`, it is possible to integrate the operator into Flux and use its Tracker module to perform backpropagation through all layers.

The example in Listing 6.2 uses a small two-layer network consisting of linearized Born modeling, followed by a fully connected layer. The loss function is defined as the mean squared error between the network output and a reference vector `y`. Since I overloaded the gradient macro for the Jacobian, derivatives of this network with respect to the weights and input vector can now be computed with the Flux tracker module. During backpropagation, Flux's AD computes the matrix-vector product of the adjoint Jacobian with the incoming data residual (by interfacing Devito under the hood) and propagates the result through the remaining Flux layer(s).

Integrating nonlinear modeling operators and parameteric layers into Flux, i.e. layers for which derivatives are computed with respect to the input as well as additional parameters, requires some extra steps but generally follows the same strategy as for linear operators. Namely, I define customized Julia types for nonlinear modeling layers and overload Flux's functions for evaluating forward and backward passes using my custom JUDI operators. An example for integrating a nonlinear forward modeling JUDI operator into a shallow CNN is shown in Listing 6.3. The network consists of two convolutional layers, with a nonlinear forward modeling layer $\mathcal{F}$ in-between them. As before, I define a loss function using Flux utilities and compute derivatives with respect to various parameters, such as the squared slowness vector `m`. Once again, gradients of layers containing JUDI operators are computed using the corresponding adjoints or JUDI gradients, instead of Flux's automatic differentiation.

```
# Nonlinear JUDI modeling operator
model = Model(n, d, o, m)
F = judiModeling(info, model, rec_geometry, src_geometry)

# Network layers
ℱ = ForwardModel(F, q)
conv1 = Conv((3, 3), 1=>1, pad=1, stride=1)
conv2 = Conv((3, 3), 1=>1, pad=1, stride=1)

# Network and loss
predict(x) = conv2(ℱ(conv1(x)))
loss(x, y) = Flux.mse(predict(x), y)

# Compute gradient w/ Flux
gs = Tracker.gradient(() -> loss(x, y), params(m))
gs[m]   # evalute gradient w.r.t. m
```

Listing 6.3: An example network combining Flux convolutional layers with a nonlinear JUDI modeling operator. Once again, derivatives of the modeling layer are implemented through JUDI operators only, but functions for evaluating the network are overloaded such that the modeling layer can be treated as a conventional Flux layer.

6.1.2 Example applications

A possible application of the ideas presented here are loop unrolling techniques for seismic imaging. Loop unrolled gradient descent and learned optimization algorithms, such as the learned primal dual reconstruction [2], are a class of (convolutional) neurals networks whose architectures are inspired by unrolled iterative optimization algorithms [10, 1, 11]. These networks follow the general structure of gradient-based optimization algorithms, in which gradients are augmented by additional neural network layers. As such, these networks are closely related to residual networks [12].

To demonstrate a possible application of my JUDI extension for Flux, I apply the loop unrolled network architecture proposed in [1] to seismic imaging. Every iteration of the unrolled algorithm consists of computing the (conventional) gradient of the LS-RTM objective function `g`, using the forward and adjoint linearized Born scattering operator. Afterwards, the gradient is concatenated along the channel dimension with the current estimate of the image `x` and a memory term `s`, both of which are initialized with Gaussian noise. The 4D tensor containing `g`, `x` and `s` (as separate channels), is then propagated through three convolutional layers with ReLU activations functions and batch normalization. Afterwards, the first channel of this output is used to update the seismic image `x` of the current iteration. The update `dx` accordingly represents the neural network augmented gradient. The output of the full network is the predicted seismic image and during training the weights of the convolutional layers are updated by minimizing its mismatch with true seismic images. The Julia code of the network in shown in Listing 6.4.

I train the network from Listing 6.4 on a small test data set consisting of $2,000$ 2D seismic images and the corresponding observed data. To reduce the number of PDE solves for computing the gradient of the LS-RTM objective function, the individual shot records of each image are summed with random weights, which yields a so-called *simultaneous* shot record [e.g 13, 14, 15, 16]. Instead of computing the LS-RTM gradient for each source

```
function network(d_obs, n; maxiter=10)
    x = randn(Float32, n[1], n[2], 1, 1)
    nx, ny, nc, nb = size(x)
    s = randn(Float32, nx, ny, 5, nb)   # memory term
    for j=1:maxiter
        g = adjoint(J)*(J*vec(x) - d_obs)
        g = reshape(Flux.normalise(g), nx, ny, 1, 1)
        u = cat(x, g, s, dims=3)
        u = batch1(conv1(u))
        u = batch2(conv2(u))
        u = batch3(conv3(u))
        s = relu.(u[:, :, 2:6, :])
        dx = u[:, :, 1:1, :]
        x += dx
    end
    return vec(x)
end
```

Listing 6.4: Example of a physics-augmented neural network for seismic imaging. The network consists of 10 iterations of an unrolled gradient descent algorithm, in which the conventional LS-RTM gradient `g` is augmented through additional convolutions layers. The input into the network is the observed seismic data and the ouput is the predicted image.

location separately and summing all gradients, we backpropagate the superposition of all shot records at the same time. This requires only 2 instead of $2n_s$PDE solves per gradient computation, but leads to noisy images (n_s being the number of shot records per image). Figure 6.1a shows an example of a noisy RTM image, which is obtained by migrating the corresponding simultaneous shot record (i.e. through multiplication with the adjoint Born operator). Figure 6.1b shows the imaging result that is obtained by performing 10 iterations of conventional gradient descent using the simultaneous data, which leads to a visually improved image in comparison to RTM, but the resulting image is still noisy. Figure 6.1c shows the output of the loop unrolled gradient descent algorithm (Listing 6.4), after training the network for $2,000$ iterations on the training data set (in which the seismic image from Figure 6.1 is not contained). The result is visually the closest to the true image (Figure 6.1d), but some residual artifacts remain.

The example presented here is not necessarily a realistic application of deep learning to seismic imaging, as the training process requires pairs of observed data and true im-

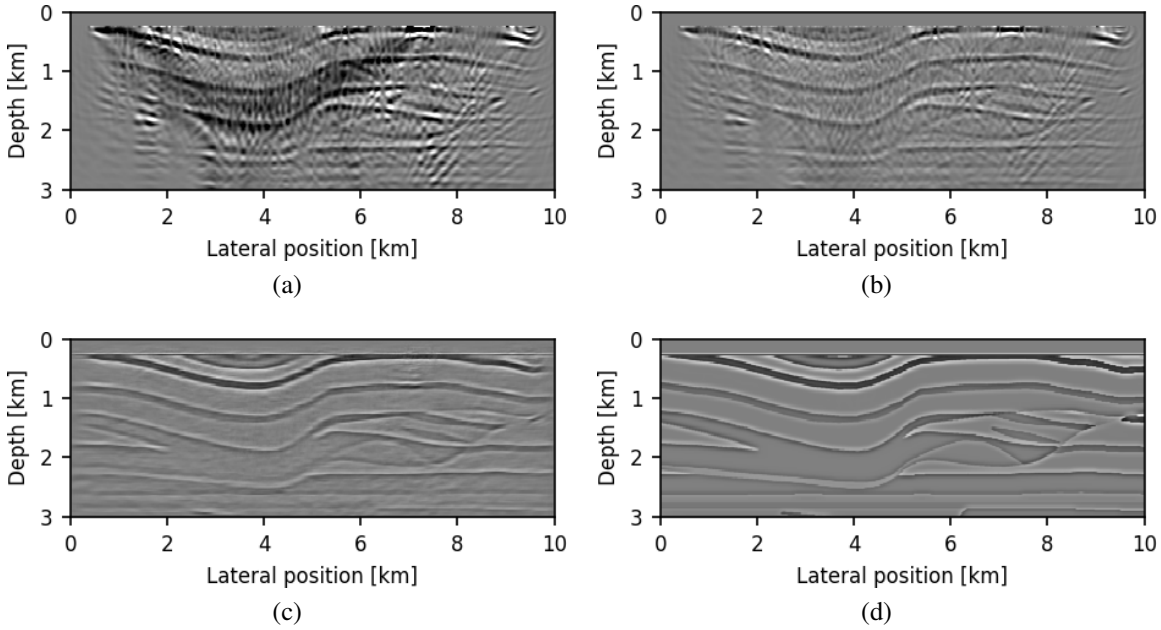


Figure 6.1: Results of various seismic imaging approaches using simultaneous shot records. Figure (a) and (b) are the RTM and LS-RTM image after 10 iterations of (conventional) gradient descent. Figure (c) is the image that is computed with the physics-augmented neural network, consisting of 10 iterations of the loop unrolled gradient descent algorithm from Listing 6.4. Figure (d) is the true image.

ages, of which the latter are not available in practice as the true subsurface is obviously unknown. However, the intention behind this example is to showcase the possibility to augment model-driven approaches based on PDEs with neural networks. Notably, the network in Listing 6.4 is not just a simple image denoiser based on a black-box CNN that learns a map from a noisy to the true image, but it represents a physics-driven network with combinations of Born modeling and convolutions. Future research directions therefore include possible applications of physics-augmented deep neural networks to seismic inverse problems, where true earth models and images are not available. Further research directions include the extension of my JUDI-Flux interface (e.g. nested derivatives) or the implementation of similar Python interfaces directly between Devito and Tensorflow/PyTorch. To my knowledge, the approach presented here is the first successful attempt to integrate time-stepping codes for wave equation based modeling into deep learning libraries.

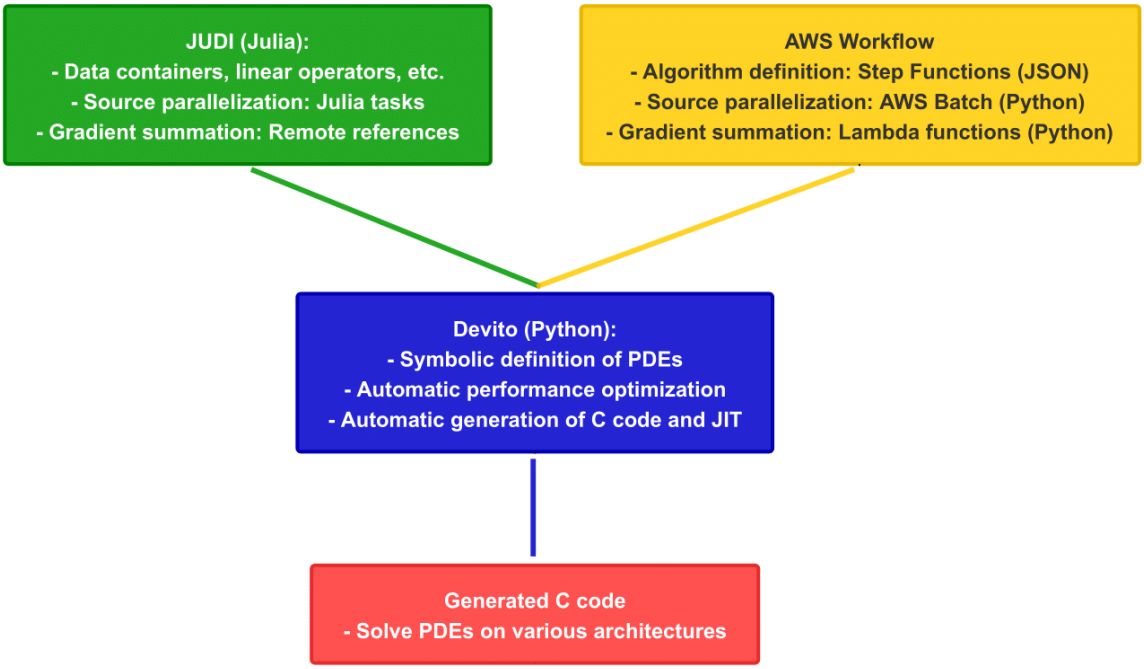


Figure 6.2: Current software structure of JUDI and the serverless seismic imaging workflow on AWS. Whereas JUDI provides a single interface for implementing parallel algorithms for seismic inverse problem based on abstract linear operators, the AWS workflow consists of a number of separate components in various programming languages.

6.2 Unified cloud interfaces and integration with JUDI

In my Julia framework, seismic modeling operators are exposed as matrix-free linear operators and inherently include the parallelization over multiple source locations. If a modeling operator represents an experiment with multiple source locations and involves solving a multiple PDEs, the implementation of the matrix-vector product first calls the parallel instance of the corresponding modeling function and distributes the workload to the available workers (see chapter 2.2.2). As discussed in chapter 2, the distribution of an embarrassingly parallel workload in Julia is based on tasks and remote references. The master process creates one task per PDE to be solved, distributes the tasks on a first-come first-served basis to the available workers and receives remote references to the results.

In the serverless seismic imaging workflow for the cloud that was presented in chapter 4, parallelization through linear operators is currently not supported, as the distribu-

tion of embarrassingly parallel worksloads is based on AWS/Azure Batch. Parallel batch jobs are created and submitted by Lambda/Azure functions, both of which currently only support Python, but not Julia. Individual batch jobs are executed as Docker containers, inside of which it is possible to use serial JUDI operators only. Similarly, the collection of gradients is implemented in Python as well, as the reduction operations are also based on Lambda/Azure functions. Finally, iterative optimization algorithms in my workflow are expressed as a collection of AWS Step Function states and are implemented in the JavaScript Object Notation (JSON).

Overall, this leads to two distinct user interfaces for conventional cluster environments and the cloud (Figure 6.2). On AWS and Azure, algorithms based on the serverless architecture cannot be implemented as single programs and require managing multiple individual components in different programming languages. This makes maintaining and extending the current framework challenging, as it requires that users are familiar with various cloud services (AWS Batch, Lambda, Step Functions) and understand how these components exactly interact with each other. Future research therefore may address the question if components of my serverless seismic imaging workflow can be abstracted and integrated into a unified framework, such as JUDI. Specifically, this would involve exposing functionalities like the event-driven gradient summation as Julia functions and their integration into JUDI's linear operators. Additionally, the development of various backends for the different cloud providers (AWS, Azure, GCP) is required in order to provide portability and the flexibility to switch platforms.

REFERENCES

[1] J. Adler and O. Öktem, "Solving ill-posed inverse problems using iterative deep neural networks," *Inverse Problems*, vol. 33, no. 12, p. 124 007, Nov. 2017.

[2] J. Adler and O. Öktem, "Learned primal-dual reconstruction," *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Medical Imaging*, vol. 37, no. 6, pp. 1322–1332, Jun. 2018.

[3] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, 2018, pp. 6571–6583.

[4] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, *Diffeqflux.jl - A Julia library for neural differential equations*, `http://arxiv.org/abs/1902.02376`, Computing Research Repository (arXiv CoRR, 2019. (visited on 09/28/2019).

[5] A. Richardson, *Seismic full-waveform inversion using deep learning tools and techniques*, `https://arxiv.org/abs/1801.07232`, arXiv e-prints, 2018. (visited on 09/09/2018).

[6] J. Sun, Z. Niu, K. A. Innanen, J. Li, and D. O. Trad, "A theory-guided deep learning formulation of seismic waveform inversion," in *89th Annual International Meeting, SEG, Expanded Abstracts*. 2019, pp. 2343–2347.

[7] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *Association for Computing Machinery (ACM) Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 135–151, 2002.

[8] S. Winograd, "On computing the discrete Fourier transform," *Mathematics of Computation*, vol. 32, no. 141, pp. 175–199, 1978.

[9] M. Innes, "Flux: Elegant machine learning with Julia," *Journal of Open Source Software (JOSS)*, vol. 3, no. 25, p. 602, 2018.

[10] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas, "Learning to learn by gradient descent by gradient descent," in *Advances in Neural Information Processing Systems*, 2016, pp. 3981–3989.

[11] C. Metzler, A. Mousavi, and R. Baraniuk, "Learned D-AMP: Principled neural network based compressive image recovery," in *Advances in Neural Information Processing Systems*, 2017, pp. 1772–1783.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on Computer Bision and Pattern Recognition*, 2016, pp. 770–778.

[13] L. A. Romero, D. C. Ghiglia, C. C. Ober, and S. A. Morton, "Phase encoding of shot records in prestack migration," *Geophysics*, vol. 65, no. 2, pp. 426–436, 2000.

[14] Y. Tang and B. Biondi, "Least-squares migration/inversion of blended data," *79th Annual International Meeting, SEG, Expanded Abstracts*, pp. 2859–2863, 2009.

[15] J. R. Krebs, J. E. Anderson, D. Hinkley, R. Neelamani, S. Lee, A. Baumstein, and M.-D. Lacasse, "Fast full-wavefield seismic inversion using encoded sources," *Geophysics*, vol. 74, no. 6, WCC177–WCC188, 2009.

[16] E. Haber, M. Chung, and F. Herrmann, "An effective method for parameter estimation with PDE constraints with multiple right-hand sides," *Society for Industrial and Applied Mathematics (SIAM) Journal on Optimization*, vol. 22, no. 3, pp. 739–757, 2012.

Appendices

APPENDIX A

A.1 Setting up wave equations with Devito

Devito is a Python domain-specific language for discretizing partial-differential equations and automatically generating optimized C code for solving them. Devito is built around symbolic functions for velocity models and (time-dependent) wavefields from which forward and adjoint wave equations can be symbolically defined. For example, we can set up a `model` structure for a two- or three-dimensional velocity model `v`, with a specified origin, grid spacing and number of absorbing boundary points `nbpml` as follows:

```
model = Model(vp=v, origin=(0,0), shape=(101,101),
 ↪  spacing=(10,10), nbpml=40)
```

Wavefields are defined as `TimeFunction` objects and are created for a specified time- and space order of their associated finite-difference derivatives:

```
u = TimeFunction(name="u", grid=model.grid, time_order=2,
 ↪  space_order=2,save=False, time_dim=nt)
```

Spatial and temporal derivatives of the wavefield `u` can be accessed via the shorthand expressions `u.dt` (first temporal derivative), `u.dt2` (second temporal derivative), `u.dx` (first spatial derivative in `x` direction) or `u.laplace` (sum of second spatial derivatives). These expressions allow us to symbolically define the acoustic wave equation with a damping term:

```
pde = model.m * u.dt2 - u.laplace + model.damp * u.dt
stencil = Eq(u.forward, solve(pde, u.forward)[0])
```

The second line rearranges the `pde` expression so that we obtain an update rule for the wavefield at the next time step `u.forward` within the forward time loop. By default, Dirichlet boundary conditions are used for this expression, but other boundary conditions can be implemented symbolically as well (e.g. Neumann). Furthermore, Devito provides the possibility to add a source function to our PDE and to sample the wavefield at receiver positions. For example, we can define a one-dimensional Ricker wavelet for a given peak frequency `f0`, which is injected into the model at some specified source coordinate. We first set up the wavelet and then inject it into the updated wavefield:

```
src = RickerSource(name="src", grid=model.grid, f0=f0,
↪  time=time,coordinate=src_coords)
src_term = src.inject(field=u.forward, expr=src * dt**2
↪  /model.m,offset=model.nbpml)
```

Receivers for given coordinates are set up in a similar fashion, but instead of injecting, we sample the wavefield and interpolate it to the receiver locations:

```
rec = Receiver(name="rec", npoint=101, nt=nt,
↪  grid=model.grid,coordinates=rec_coords)
rec_term = rec.interpolate(u, offset=model.nbpml)
```

To generate the forward modeling operator, we add the source and receiver terms to our `stencil` expression and pass it to Devito's `Operator` function, which generates optimized stencil code with a time-stepping loop for solving the wave equation. We can then run the generated C code for a specified length and time step with:

```
op_fwd = Operator([stencil] + src_term + rec_term)  #
↪  generate code
op_fwd(time=nt, dt=model.critical_dt)   # run it
```

The instructions presented here are a short summary of a detailled tutorial series on setting up forward and adjoint acoustic wave equations that has been published in the Leading Edge [1, 2]. The tutorials also provide details on implementing absorbing boundary conditions for simulating infinite domains. In JUDI, the wave equations are set up following these tutorials, and the code can be found and modified in `~/.julia/dev/JUDI/src/Python/JAcoustic_codegen.py`.

A.2 Relationship between impedance imaging and inverse scattering

A comparison of reverse time-migration with the linearized inverse scattering imaging condition (ISIC) [3, 4] and seismic imaging with the acoustic impedance [5], reveals that the respective sensitivity kernels at equivalent. The sensitivity kernel (i.e. the image) for the acoustic impedance $K_Z(\mathbf{x})$ is defined in [5] as the sum of the sensitivity kernels of the spatially varying bulk modulus $\boldsymbol{\kappa}$ and density $\boldsymbol{\rho}$:

$$K_Z(\mathbf{x}) = K_\kappa(\mathbf{x}) + K_\rho(\mathbf{x}), \tag{A.1}$$

where the sensitivity kernel is defined as:

$$K_\kappa(\mathbf{x}) = -\frac{1}{\boldsymbol{\kappa}} \sum_{i=1}^{n_t} \text{diag}\big(\dot{\mathbf{u}}_i\big)\dot{\mathbf{v}}_i, \tag{A.2}$$

and $\dot{\mathbf{u}}_i, \dot{\mathbf{v}}_i$ are first time-derivatives of forward and adjoint wavefields. The sensitivity kernel for the density is given by:

$$K_\rho(\mathbf{x}) = \frac{1}{\boldsymbol{\rho}} \sum_{i=1}^{n_t} \text{diag}\big(\nabla \mathbf{u}_i\big)\nabla \mathbf{v}_i, \tag{A.3}$$

where the second term denotes the pointwise products of the spatial derivatives of forward and adjoint wavefields. Combining the two equations and substituting the bulk modules by

the velocity and density yields:

$$K_Z(\mathbf{x}) = -\sum_{i=1}^{n_t}\left[\frac{1}{\boldsymbol{\rho}\mathbf{v}^2}\text{diag}\big(\dot{\mathbf{u}}_i\big)\dot{\mathbf{v}}_i - \frac{1}{\boldsymbol{\rho}}\text{diag}\big(\nabla\mathbf{u}_i\big)\nabla\mathbf{v}_i\right]. \tag{A.4}$$

A comparison of this expression with equation 3.9 reveals that the impedance kernel is equivalent to the linearized inverse scattering imaging condition for $\rho(\mathbf{x}) = 1$. This is independent of whether we consider the frequency-domain formulation of ISIC [3] or its time-domain equivalent [4]. Multiplication of the frequency-domain source wavefield with ω^2 corresponds to a second time derivative of the forward time-domain wavefield, or to first time derivatives of both forward and adjoint wavefields.

A.3 Physical interpretation of the linearized Bregman method for seismic imaging

The linearized Bregman method used in chapter 3 to solve the ℓ_1-minimization problem in equation 3.13 is a specialized case of a broader class of optimization problems for solving convex, but potentially non-differentiable objective functions with (in-)equality constraints. Namely, the linearized Bregman method is a simplification of the more general Bregman iterative regularization method, and can be derived by linearizing the quadratic data fidelity term in classic Bregman iterations [6].

The advantage of the linearized Bregman algorithm in comparison to the more general Bregman iterative regularization or the augmented Lagrangian method, is that every iteration involves only two matrix-vector products; $\mathbf{J}\mathbf{x}$ and $\mathbf{J}^\top\big(\mathbf{d}_{\text{pred}} - \mathbf{d}_{\text{obs}}\big)$. In the case of seismic imaging, where $\mathbf{J}$ is the linearized Born scattering operator, these matrix-vector products correspond precisely to linearized Born modeling (Step 4) and reverse-time migration (Step 5). The Born scattering operator for a full seismic survey is overdetermined, i.e. there are (significantly) more observed data points than coefficients in the seismic image. However, because working with the full Born scattering operator in each iteration is prohibitively expensive, as it involves the demigration/migration of all shots, we can work

with random subsets of frequencies and shots (or simultaneous shots) [7]. This has the effect of turning the overdetermined problem, into an underdetermined problem, but, as demonstrated in Figure A.1, also leads to a noisy image (variable $\mathbf{z}$ in the algorithm). By using sparsity-promoting minimization, it is possible to remove the noise and recover the true image in the subsequent iterations. However, in order for sparsity promotion to be successful, it is crucial that the noise is in fact incoherent and does not contain any aliases. By choosing the subsets of frequencies (and shots) randomly in each iteration, we guarantee that the image contains only incoherent noise and no aliases or wrap-around effects, as would be the case for periodic subsampling.

Algorithm A.1 A simplified version of the linearized Bregman method from Algorithm 3.1 without preconditioners.

1. Initialize $\mathbf{x}_1 = \mathbf{0}$, $\mathbf{z}_1 = \mathbf{0}$, q, λ, batch sizes $\hat{n}_s \ll n_s$ and $\hat{n}_f \ll n_f$
2. **for** $i = 1, ..., n$
3. Select subset of shots and frequencies $\mathcal{S} = (\int_{\text{shot}}, \int_{\text{freq}}), |\int_{\text{shot}}| = \hat{n}_s, |\int_{\text{freq}}| = \hat{n}_f$
4. $\bar{\mathbf{d}}_{\mathcal{S}}^{\text{pred}} = \mathbf{J}_{\mathcal{S}}\mathbf{x}$
5. $\bar{\mathbf{g}}_{\mathcal{S}} = \mathbf{J}_{\mathcal{S}}^{\top}\left(\bar{\mathbf{d}}_{\mathcal{S}}^{\text{pred}} - \bar{\mathbf{d}}_{\mathcal{S}}^{\text{obs}}\right)$
6. $\mathbf{z}_{i+1} = \mathbf{z}_i - t_i\bar{\mathbf{g}}_{\mathcal{S}}$
7. $\mathbf{x}_{i+1} = \mathbf{C}^{\top} S_{\lambda}(\mathbf{C}\mathbf{z}_{i+1})$
8. **end**

with $S_{\lambda}(\mathbf{C}\mathbf{z}) = \text{sign}(\mathbf{C}\mathbf{z}) \cdot \max(0, |\mathbf{C}\mathbf{z}| - \lambda)$

In step 7 of the algorithm, we compute the curvelet transform of the noisy image $\mathbf{z}_i$, since we want to promote sparsity of the image in the curvelet domain. This is followed by applying the soft-thresholding function to the noisy coefficients, which effectively sets all coefficients smaller than λ to zero and shrinks the magnitude of the remaining values by λ. In the early iterations, this sets not only the noise, but also coefficients of reflectors to zero. During the subsequent iterations, the amplitude of the reflector coefficients is continuously increased, such that toward the final iterations, the soft thresholding function removes (ideally) only the noise. This is illustrated in Figure A.1, which shows the variables $\mathbf{z}_1$ and $\mathbf{x}_1$ in comparison to $\mathbf{z}_{20}$ and $\mathbf{x}_{20}$, i.e. both variables during the first and final iteration.

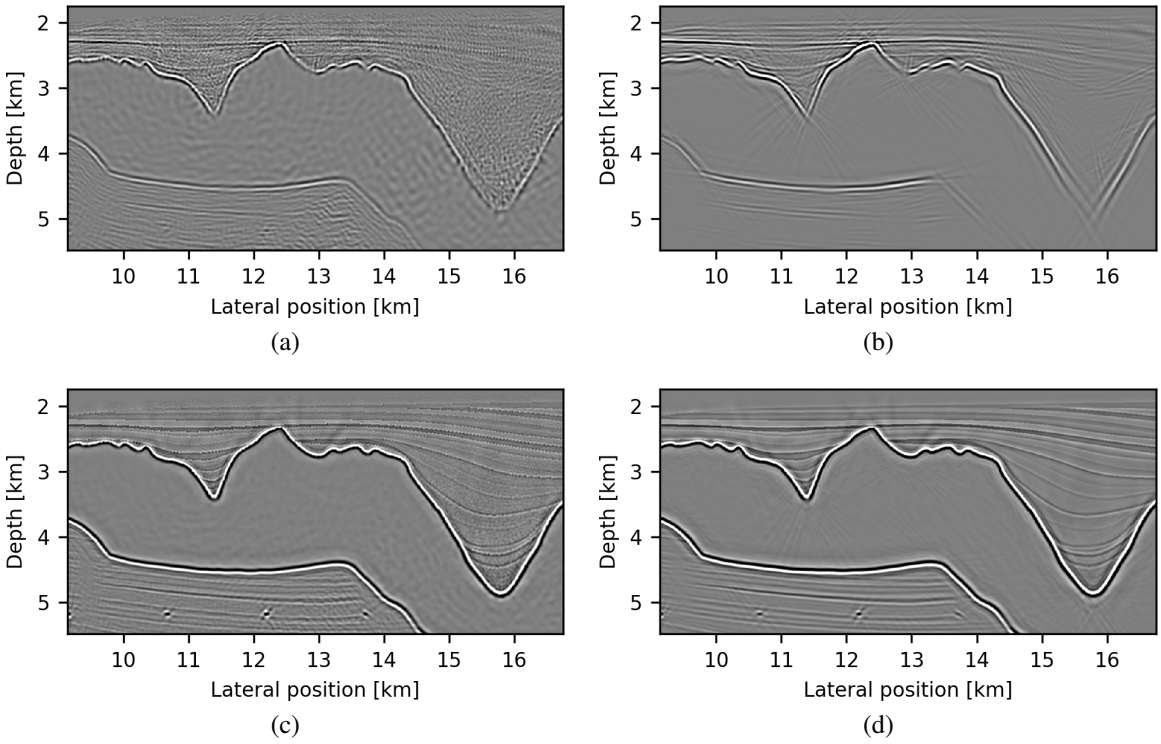


Figure A.1: After the first iteration of the linearized Bregman method, the dual variable $\mathbf{z}$ (a) is a noisy version of the seismic image. The sparse primal variable $\mathbf{x}$ (b) is obtained by soft-thresholding of the curvelet coefficients of (a). In the early iterations, $\mathbf{x}$ contains only reflectors with the largest (curvelet) coefficients, but the smaller coefficients re-enter the solution in the subsequent iterations. After the final iteration, all reflectors have been restored in the primal variable (d), while the dual variable (c) is still noisy (but less than after the initial iteration).

A.4 Utilized hardware and software

A.4.1 Chapter 3

The timings shown in Figure 3.8 were computed for the Sigsbee 2A velocity model with a grid spacing of 7.62 m, which corresponds to $1,201 \times 3,201$ grid points. The time stepping interval according to the CFL condition is 0.71 ms, resulting in $14,095$ time steps for 10 seconds modeling time. The timings for the BP model (Figure 3.16b) were computing using a grid spacing of 6.25 m ($1,911 \times 10,789$ grid points) and 12 seconds modeling time with a time stepping interval of 0.548 ms ($21,898$ time steps).

All timings were computed with an Intel Xeon E5 v2 processors (2.8 GHz) with 10

cores and 128 GB RAM. Each shown time measurement is the smallest runtime of three individual runs. The examples were computed using 10 threads, in which each thread is pinned to a specific core (thread pinning).

The following software was used for the timings and numerical case studies: Julia (v0.6.3), JUDI (v0.2.1:dft-paper), Python (v3.6.5), Devito (v3.2.0:dft-paper), Intel compiler (v16.0.3).

A.4.2 Chapter 4

Table A.1 provides an overview of the AWS EC2 instances used in our performance analysis, including their respective CPU architectures. EC2 instances of a fixed instance type (such as `r5` or `c5n`) have the same architecture for different sizes (e.g. `2xlarge`, `4xlarge`), as those instances run on the same hardware.

Table A.1: Architectures of compute instances used in our performance analysis on AWS and Optimum.

Instance	Intel Xeon Architecture	vCPUs	RAM (GB)
m4.4xlarge	E5-2686 v4 @ 2.30GHz	16	64
r5.12xlarge	Platinum 8175M @ 2.50 GHz	48	384
r5.24xlarge	Platinum 8175M @ 2.50 GHz	96	768
c5n.9xlarge	Platinum 8124M @ 3.00 GHz	36	384
c5n.12xlarge	Platinum 8142M @ 3.00 GHz	72	768
r5.metal	Platinum 8175M @ 2.50 GHz	96	768
Optimum	E5-2680 v2 @ 2.80GHz	20	256

A.5 Model and data dimensions

Table A.2 lists the dimensions of the BP 2004 model and the corresponding seismic data set that was used in our performance analysis. Both the model and data set are publicly available from the society of exploration geophysicists [8].

Table A.2: Parameters of the BP 2004 velocity benchmark model and the corresponding seismic data set.

Grid dimensions	$1,911 \times 10,789$
Grid spacing [m]	6.25×6.25
Domain size [km]	11.94×67.43
Number of seismic source n_s	$1,348$
Propagation time [s]	12
Number of time steps	$21,889$
Dimensions of each $\mathbf{d_i}$ (reshaped to 2D array)	$2,001 \times 1,201$
Dominant frequency of source [Hz]	20

REFERENCES

[1] M. Louboutin, P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, "Full-waveform inversion, Part 1: Forward modeling," *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, 2017.

[2] ——, "Full-waveform inversion, Part 2: Adjoint modeling," *The Leading Edge*, vol. 37, no. 1, pp. 69–72, 2018.

[3] T. J. Op't Root, C. C. Stolk, and M. V. de Hoop, "Linearized inverse scattering based on seismic reverse time migration," *Journal de Mathematiques Pures et Appliquees*, vol. 98, no. 2, pp. 211–238, 2012.

[4] N. D. Whitmore and S. Crawley, "Applications of RTM inverse scattering imaging conditions," *82nd Annual International Meeting, SEG, Expanded Abstracts*, pp. 1–6, 2012.

[5] H. Douma, D. Yingst, I. Vasconcelos, and J. Tromp, "On the connection between artifact filtering in reverse-time migration and adjoint tomography," *Geophysics*, vol. 75, no. 6, S219–S223, 2010.

[6] W. Yin, S. Osher, D. Goldfarb, and J. Darbon, "Bregman iterative algorithms for ℓ_1-minimization with applications to compressed sensing," *Society for Industrial and Applied Mathematics (SIAM) Journal on Imaging sciences*, vol. 1, no. 1, pp. 143–168, 2008.

[7] D. A. Lorenz, S. Wenger, F. Schöpfer, and M. Magnor, "A sparse Kaczmarz solver and a Linearized Bregman method for online compressed sensing," *Institute of Electrical and Electronics Engineers (IEEE): International Conference on Image Processing*, pp. 1347–1351, 2014.

[8] *2004 BP velocity estimation benchmark model*, `https://wiki.seg.org/wiki/2004_BP_velocity_estimation_benchmark_model`, 2019. (visited on 08/01/2019).

APPENDIX B

B.1 Permissions to use copyrighted material

B.1.1 Chapter 2

The content of chapter 2 was published as a technical article in *Geophysics*, under the title ”A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia”:

- P. A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G. J. Gorman and F. J. Herrmann, ”A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia”, *Geophysics*, vol. 84, no. 3 , pp. F57-F71, 2019.
- DOI: https://doi.org/10.1190/geo2018-0174.1
- Copyright © 2019 Geophysics
- The author retains the right to reuse all or part of the work in a thesis or dissertation, as stated in the Copyright Agreement.

Manuscript Number: GEO-2018-0174

Geophysics Transfer of Copyright Agreement

Agreement must be signed by lead or corresponding author and returned to SEG Business Office before article receives final acceptance.

Article title: A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia

Names of all authors: Philipp A. Witte, Mathias Louboutin, Navjot Kukreja, Fabio Luporini, Michael Lange, Gerard J. Gorman, Felix J. Herrmann

For and in consideration of the potential publication of the article listed above (the "Work") by the Society of Exploration Geophysicists ("SEG"), I, the undersigned, as lead and/or corresponding author and/or rightsholders and acting on behalf of all authors and/or owners of copyright and other rights in the Work ("Authors"), hereby transfer, assign, and convey all right, title, interest, and worldwide copyright in the Work to SEG, effective if and when the Work is accepted for publication, subject only to limitations expressed in this Transfer of Copyright (the "Agreement"). I warrant that I am authorized and empowered to represent all Authors with respect to the Agreement, which is executed jointly and severally by Authors if copyright transfer is from multiple individuals or entities. The Work includes the article and all material to be published within and with the article in any and all media, including but not limited to tables, figures, graphs, source code, movies, and other multimedia. Responses to discussions of the Work, errata, and other similar material directly related to the Work that may arise subsequent to publication also are considered part of the Work. In the event the Work incorporates copyrighted material of others, Authors warrant that all required permissions or releases have been secured. Authors agree to indemnify, defend, and hold SEG, its directors, officers, employees, and agents harmless against any claims to the contrary.

Authors shall retain the following royalty-free rights:

1. All proprietary rights in the Work that are not transferred to SEG in the Agreement, including the right to any patentable subject matter that may be contained in the Work
2. The right to reproduce and distribute part or all of the Work, including figures, drawings, tables, and abstracts of the Work, with proper attribution and copyright acknowledgment, in connection with Authors' teaching and technical collaborations
3. The right to make oral presentation of the same or similar information as that contained in the Work provided acknowledgment is made of SEG copyright ownership and publication status
4. The right to post a final accepted version of the manuscript or the final SEG-formatted version on Authors' personal Web sites (not including social-network sites used for article sharing such as ResearchGate), employers' Web sites, or in institutional repositories operated and controlled exclusively by Authors' employers provided that (a) the *Geophysics*-accepted or *Geophysics*-published version is presented without modification unless modification is noted and fully described; (b) SEG copyright notice and a full citation appear with the paper; (c) a live link to the version of record in the SEG Digital Library using a Digital Object Identifier (DOI) permalink is provided; (d) the posting is noncommercial in nature, and the paper is made available to users without charge; (e) notice is provided that use is subject to SEG terms of use and conditions; and (f) a posting of the Work in an institutional repository neither carries nor is implied as carrying a license in conflict with SEG copyright and terms of use
5. The right to prepare and hold copyright in derivative publications based on the Work provided that the derivative work is published subsequent to the official date of the Work's publication by SEG and the Work is cited
6. The right to post to their own Web sites a preprint (a version prior to *Geophysics*' accepted and published versions) of the Work provided that the posting is accompanied by prominent notice that the Work is under review for publication in *Geophysics* and that upon publication by SEG in any form, the preprint is taken offline and replaced with either (a) a full citation of the published Work that may include the article abstract, plus a DOI permalink to the version of record; (b) the final published version, subject to SEG conditions for such posting; or (c) a modification of the Work with notification stating the work is a modification of the SEG version, also subject to SEG conditions for such posting; and that if the Work is rejected for publication by SEG, notice of its under-review status be removed
7. The right to reuse all or part of the Work in a thesis or dissertation that the Author writes and is required to submit to satisfy criteria of degree-granting institutions, with full citation of the Work, its copyright status, and a DOI permalink to the version of record
8. The nonexclusive right, after publication by SEG, to republish and distribute print versions of the Work or excerpts therefrom without obtaining permission from SEG, provided that (a) the paper is not republished in a journal, book, or collection of conference abstracts or proceedings; and (b) no fee is charged for the printed versions. Permission must be obtained from SEG for other republication of the Work.

SEG may republish the Work or portions thereof in any future SEG publication or compilation in any form and in any language, and SEG retains exclusive right to license third parties to do the same.

This Agreement entitles Authors (in the case of a Work Made for Hire, the employer) to retain all rights not transferred, assigned, or conveyed to SEG in the Agreement. Authors confirm that the Work has not been published previously elsewhere, nor is it under consideration by any other publisher, that the Work does not infringe any copyright or invade any right of privacy or publicity, and that Authors have the full power to enter into this Agreement and to make the grants contained herein. Authors warrant that they have complied and will comply with Ethical Guidelines for SEG Publications and that they agree to the provisions of these guidelines. This Agreement shall be binding on Authors' heirs, executors, administrators, and assigns, and shall be construed in accordance with the laws of the State of Oklahoma, United States of America.

IN WITNESS WHEREOF, I have executed this Transfer of Copyright on this 1 day of September, 2018.

Philipp A. Witte
Name of author (print or type)

Philipp Witte
Signature

Georgia Institute of Technology
Company for which work was performed (if applicable)

Professor Felix J. Herrmann, Advisor
Authorized by/Title

SIGN HERE IF U.S. GOVERNMENT EMPLOYED ALL AUTHORS WHEN WORK WAS PREPARED.

I certify that the article named above was prepared solely by a U.S. government employee(s) as part of his/her (their) official duties and therefore legally cannot by copyrighted. Authors agree to all other terms of this Agreement.

Name (print or type) Date

Signature

B.1.2 Chapter 3

The content of chapter 3 was published as a technical article in *Geophysics*, under the title ”Compressive least-squares migration with on-the-fly Fourier transforms”:

- P. A. Witte, M. Louboutin, F. Luporini, G. J. Gorman and F. J. Herrmann, ”Compressive least-squares migration with on-the-fly Fourier transforms”, *Geophysics*, vol. 84, no. 5 , pp. R655-R672, 2019.
- DOI: https://doi.org/10.1190/geo2018-0490.1
- Copyright © 2019 Geophysics
- The author retains the right to reuse all or part of the work in a thesis or dissertation, as stated in the Copyright Agreement.

Manuscript Number: GEO-2018-049

Geophysics Transfer of Copyright Agreement

Agreement must be signed by lead or corresponding author and returned to SEG Business Office before article receives final acceptance.

Article title: Compressive least-squares migration with on-the-fly Fourier transforms

Names of all authors: Philipp A. Witte, Mathias Louboutin, Fabio Luporini, Gerard J. Gorman, Felix J. Herrmann

For and in consideration of the potential publication of the article listed above (the "Work") by the Society of Exploration Geophysicists ("SEG"), I, the undersigned, as lead and/or corresponding author and/or rightsholders and acting on behalf of all authors and/or owners of copyright and other rights in the Work ("Authors"), hereby transfer, assign, and convey all right, title, interest, and worldwide copyright in the Work to SEG, effective if and when the Work is accepted for publication, subject only to limitations expressed in this Transfer of Copyright (the "Agreement"). I warrant that I am authorized and empowered to represent all Authors with respect to the Agreement, which is executed jointly and severally by Authors if copyright transfer is from multiple individuals or entities. The Work includes the article and all material to be published within and with the article in any and all media, including but not limited to tables, figures, graphs, source code, movies, and other multimedia. Responses to discussions of the Work, errata, and other similar material directly related to the Work that may arise subsequent to publication also are considered part of the Work. In the event the Work incorporates copyrighted material of others, Authors warrant that all required permissions or releases have been secured. Authors agree to indemnify, defend, and hold SEG, its directors, officers, employees, and agents harmless against any claims to the contrary.

Authors shall retain the following royalty-free rights:

1. All proprietary rights in the Work that are not transferred to SEG in the Agreement, including the right to any patentable subject matter that may be contained in the Work
2. The right to reproduce and distribute part or all of the Work, including figures, drawings, tables, and abstracts of the Work, with proper attribution and copyright acknowledgment, in connection with Authors' teaching and technical collaborations
3. The right to make oral presentation of the same or similar information as that contained in the Work provided acknowledgment is made of SEG copyright ownership and publication status
4. The right to post a final accepted version of the manuscript or the final SEG-formatted version on Authors' personal Web sites (not including social-network sites used for article sharing such as ResearchGate), employers' Web sites, or in institutional repositories operated and controlled exclusively by Authors' employers provided that (a) the *Geophysics*-accepted or *Geophysics*-published version is presented without modification unless modification is noted and fully described; (b) SEG copyright notice and a full citation appear with the paper; (c) a live link to the version of record in the SEG Digital Library using a Digital Object Identifier (DOI) permalink is provided; (d) the posting is noncommercial in nature, and the paper is made available to users without charge; (e) notice is provided that use is subject to SEG terms of use and conditions; and (f) a posting of the Work in an institutional repository neither carries nor is implied as carrying a license in conflict with SEG copyright and terms of use
5. The right to prepare and hold copyright in derivative publications based on the Work provided that the derivative work is published subsequent to the official date of the Work's publication by SEG and the Work is cited
6. The right to post to their own Web sites a preprint (a version prior to *Geophysics*' accepted and published versions) of the Work provided that the posting is accompanied by prominent notice that the Work is under review for publication in *Geophysics* and that upon publication by SEG in any form, the preprint is taken offline and replaced with either (a) a full citation of the published Work that may include the article abstract, plus a DOI permalink to the version of record; (b) the final published version, subject to SEG conditions for such posting; or (c) a modification of the Work with notification stating the work is a modification of the SEG version, also subject to SEG conditions for such posting; and that if the Work is rejected for publication by SEG, notice of its under-review status be removed
7. The right to reuse all or part of the Work in a thesis or dissertation that the Author writes and is required to submit to satisfy criteria of degree-granting institutions, with full citation of the Work, its copyright status, and a DOI permalink to the version of record
8. The nonexclusive right, after publication by SEG, to republish and distribute print versions of the Work or excerpts therefrom without obtaining permission from SEG, provided that (a) the paper is not republished in a journal, book, or collection of conference abstracts or proceedings; and (b) no fee is charged for the printed versions. Permission must be obtained from SEG for other republication of the Work.

SEG may republish the Work or portions thereof in any future SEG publication or compilation in any form and in any language, and SEG retains exclusive right to license third parties to do the same.

This Agreement entitles Authors (in the case of a Work Made for Hire, the employer) to retain all rights not transferred, assigned, or conveyed to SEG in the Agreement. Authors confirm that the Work has not been published previously elsewhere, nor is it under consideration by any other publisher, that the Work does not infringe any copyright or invade any right of privacy or publicity, and that Authors have the full power to enter into this Agreement and to make the grants contained herein. Authors warrant that they have complied and will comply with Ethical Guidelines for SEG Publications and that they agree to the provisions of these guidelines. This Agreement shall be binding on Authors' heirs, executors, administrators, and assigns, and shall be construed in accordance with the laws of the State of Oklahoma, United States of America.

IN WITNESS WHEREOF, I have executed this Transfer of Copyright on this 6 day of March, 2019.

Philipp A. Witte
Name of author (print or type)

Philipp Witte
Signature

Georgia Institute of Technology
Company for which work was performed (if applicable)

Professor Felix J. Herrman, Advisor
Authorized by/Title

SIGN HERE IF U.S. GOVERNMENT EMPLOYED ALL AUTHORS WHEN WORK WAS PREPARED.

I certify that the article named above was prepared solely by a U.S. government employee(s) as part of his/her (their) official duties and therefore legally cannot by copyrighted. Authors agree to all other terms of this Agreement.

Name (print or type) Date

Signature

B.1.3 Chapter 4

The content of chapter 4 was submitted in modified form as a technical article to *IEEE Transactions on Parallel and Distributed Systems* in August 2019.

- P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage and F. J. Herrmann, ”An event-driven approach to serverless seismic imaging in the cloud”, submitted to *IEEE Transactions on Parallel and Distributed Systems*, August 2019.
- The paper is currently under review and has not been published by IEEE.
- No transfer of copyright has been signed and the copyright remains with the author.