A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia

Philipp A. Witte[*], **Mathias Louboutin**[*], **Navjot Kukreja**[†], **Fabio Luporini**[†], **Michael Lange**[‡], **Gerard J. Gorman**[†] **and Felix J. Herrmann**[*]

[*]*Georgia Institute of Technology*

School of Computational Science and Engineering

Cherry Emerson Bldg, Ferst Drive

Atlanta, GA, 30313, U.S.A.

[†]*Imperial College London,*

Department of Earth Science & Engineering

Royal School of Mines, Prince Consort Rd, Kensington

London, SW7 2BP, U.K.

[‡]*European Centre for Medium-Range Weather Forecasts*

Shinfield Rd, Reading, RG2 9AX, U.K.

(September 24, 2019)

GEO-2018-0174.R1

Running head: **A symbolic seismic inversion framework**

ABSTRACT

Writing software packages for seismic inversion is a very challenging task, since problems such as full-waveform inversion or least-squares imaging are both algorithmically and computationally demanding due to the large number of unknown parameters and the fact that waves are propagated over many wavelengths. Software frameworks therefore need to com-

bine both versatility and performance to provide geophysicists with the means and flexibility to implement complex algorithms that scale to exceedingly large 3D problems. Following these principles, we introduce the Julia Devito Inversion framework, an open-source software package in Julia for large-scale seismic modeling and inversion based on Devito, a domain-specific language compiler for automatic code generation. The framework consists of matrix-free linear operators for implementing seismic inversion algorithms that closely resemble the mathematical notation, a flexible resilient parallelization and an interface to Devito for generating optimized stencil code to solve the underlying wave equations. In comparison to many manually optimized industry codes written in low-level languages, our software is built on the idea of independent layers of abstractions and user interfaces with symbolic operators. Through a series of numerical examples, we demonstrate that this allows users to implement a series of increasingly complex algorithms for waveform inversion and imaging as simple Julia scripts that scale to large-scale 3D problems. This illustrates, that software based on the paradigms of abstract user interfaces and automatic code generation, makes it possible to manage both the complexity of algorithms and performance optimizations, thus providing a high-performance research and production framework.

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) (Tarantola, 1984; Virieux and Operto, 2009) or least-squares reverse-time migration (LS-RTM) (Nemeth et al., 1999; Tang and Biondi, 2009) 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 (Symes, 2017).

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 (Fomel et al., 2013) 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++ (Symes and Dong, 2010; Symes et al., 2011). It combines a stand-alone package for solving time-domain wave equations called iWave (Terentyev, 2009), with the Rice Vector Library (RVL) (Padula et al., 2009), 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 (Ruthotto et al., 2016) and Waveform (Da Silva and Herrmann, 2017), 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 (Nickolls et al., 2008) and OpenCL (Tompson and Schlachter, 2012). 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 (Hassanzadeh and Mosher, 1997), the RTM/FWI

framework SAVA (Koehn, 2017), a modeling and migration package by Thorbecke (2017) and a finite-element inversion framework for global seismology by Krischer et al. (2015).

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 (Abadi et al., 2016) or PyTorch (Paszke et al., 2017), 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 (Rathgeber et al., 2015) and FEniCS (Logg et al., 2012).

Based on these paradigms of domain-specific abstractions and automatic performance optimizations, we introduce a framework for seismic modeling and inversion in Julia. Bor-

rowing 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 (Lange et al., 2016; Kukreja et al., 2016; Louboutin et al., 2018a; Luporini et al., 2018) 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 (Joyner et al., 2012), 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 (Luporini et al., 2016), 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.

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 (Witte et al., 2017). The framework is implemented in Julia (Bezanson et al., 2012, 2017), 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 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 1 about here.]

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 (Claerbout, 1992). 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 (Balay et al., 2016), Trilinos (Heroux et al., 2005), or Matlab libraries such as Sparco (van den Berg et al., 2009) and SPOT (van den Berg and Friedlander, 2013). Seismic modeling and inversion frameworks that use matrix-free linear operators and data containers are RVL/iWave++ (Padula et al., 2009; Symes and Dong, 2010) or the frequency-domain framework Waveform (Da Silva and Herrmann, 2017).

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 Padula et al. (2009), 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 SeisIO.jl package (Lensink, 2017), 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 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.

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 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, Bezanson et al. (2017)). When called for more than one source/shot record, the parallel function instance of `time_modeling` is executed, which distributes the source lo-

```
# Forward and adjoint (time-reversal) modeling
d_pred = Pr*A_inv*adjoint(Ps)*q
q_ad = Ps*adjoint(A_inv)*adjoint(Pr)*(d_pred - d_obs)

# Migration/demigration
J = judiJacobian(Pr*A_inv*adjoint(Ps),q) # linearize modeling ↩
    operator
d_lin = J*dm
rtm = adjoint(J)*d_lin
```

Listing 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.

cations and data amongst the available computational resources and then calls the serial function instance with the interface to the wave equation solver (Figure 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.

Interface to the wave equation solver: Devito

The final abstraction layer of our software framework (Figure 1) is the serial instance of the `time_modeling` function, which contains the Julia interface to Devito (Lange et al., 2016). 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 (Williams et al., 2009; Louboutin et al., 2017a).

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 (Louboutin et al., 2018a; Luporini et al., 2018). 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 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) (Louboutin et al., 2017b, 2018b; Witte et al., 2018). 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 (Dagum and Menon, 1998) 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 (Louboutin et al., 2017b, 2018b).

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 (Johnson, 2017). 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.

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 (Zeng and Gullberg, 2000). 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. Unit tests in JUDI include comparisons of PDE solutions to reference codes, adjoint tests for linear operators, as well as gradient tests for non-linear objective functions (Appendix B).

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:

- 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 (Schmidt et al., 2009). 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 Virieux and Operto (2009) for a theoretical overview of FWI. Furthermore, a detailled tutorial on implementing FWI with Devito and JUDI is presented in Louboutin et al. (2017b) and Witte et al. (2018).

Full-waveform inversion

Our first numerical case study demonstrates how to implement a basic FWI example with (stochastic) gradient descent (Bottou, 2010) 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 (van Leeuwen and Herrmann, 2013) or FWI via the matched source extension (Huang et al., 2017), 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:

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

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 1 with respect to the model parameters $\mathbf{m}$ is computed using the adjoint state method (Tarantola, 1984; Plessix, 2006), also known as a reduced space method in the optimization literature (e.g., Cervantes et al.,

2000) 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 stand-alone 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 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 $\phi(\mathbf{x}) = \epsilon^2\big(\sqrt{1+\mathbf{x}^2/\epsilon^2} - 1\big)$ (Guitton and Symes, 2003; van Leeuwen et al., 2013). 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.

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 include 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.

```
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: 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.*

With our objective function in place, we can now implement a simple stochastic gradient descent algorithm in Julia (Listing 3). 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 (Bottou, 2010). 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 (Tan et al., 2016), 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 3, with 20 randomly selected shots per iteration. We use bound constraints to restrict the velocity to an interval between 1500 and 6500 m/s, while keeping the water velocity fixed at 1500 m/s. The result after 20 iterations is shown in Figure 2. 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 (Griewank and Walther, 2000; Symes, 2007; Kukreja et al., 2018), enables us

```
# 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 3: 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.

to run the same script on large-scale 3D models, as we will demonstrate in the subsequent example.

[Figure 2 about here.]

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 (Schmidt et al., 2009), 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 4 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 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.

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

```
# 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 4: 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.

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 9400 shot records with 8 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 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 1080) 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 (Griewank and Walther, 2000; Symes, 2007; Kukreja et al., 2018). A depth slice of the final result is shown in Figure 3. Apart from the minConf library, we tested interfacing the NLopt library (Johnson, 2017), which features, amongst others, limited-memory Quasi-Newton methods.

[Figure 3 about here.]

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., Tang and Biondi, 2009; Dong et al., 2012). JUDI, with its matrix-free modeling operators and data containers, is de-

signed 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 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 (Herrmann et al., 2008):

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

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 1 and the corresponding code that implements this method using JUDI is shown in Listing 5. Each iteration involves selecting a random subset of shot records and extracting the corresponding blocks of rows from the demigration operator `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 Modzelewski (2017)), the algorithm translates directly to runnable Julia code.

```
# 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 5: 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 1 to runnable Julia code.

Algorithm 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 5 itself is serial, while the parallelization happens implicitly inside `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 (Zhang et al., 2015), as shown in Algorithm 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 6). 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.

Algorithm 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 6: 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.

The Julia codes for serial and parallel SGD (Listings 5 and 6) 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 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.

[Figure 4 about here.]

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 3); thus requiring substantially

less memory (see Sirgue et al. (2010) within the context of FWI). In the adjoint time loop for migration (Algorithm 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 3 Pseudo-code for calculating frequency-domain wavefields $\hat{\mathbf{u}}_{real}$ and $\hat{\mathbf{u}}_{imag}$ 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: $\mathbf{u}_j = ...$

$\hat{\mathbf{u}}_{real} = \hat{\mathbf{u}}_{real} + \mathbf{u}_j \cos\left(2\pi f j \Delta t\right)$

$\hat{\mathbf{u}}_{imag} = \hat{\mathbf{u}}_{imag} - \mathbf{u}_j \sin\left(2\pi f j \Delta t\right)$

end

Algorithm 4 The frequency-domain gradient $\hat{\mathbf{g}}$ 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: $\mathbf{v}_j = ...$

$\hat{\mathbf{g}} = \hat{\mathbf{g}} + 4\pi^2 f^2 \mathbf{v}_j \Big(\hat{\mathbf{u}}_{real} \cos\left(2\pi f j \Delta t\right) - \hat{\mathbf{u}}_{imag} \sin\left(2\pi f j \Delta t\right) \Big)$

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 7 to our symbolic PDE

expressions for forward and adjoint modeling, that are defined in the source code of JUDI.

```
# 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 7: 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.

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 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) 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 5). For solving the sparsity-promoting LS-RTM problem with frequency subsampling, we use the linearized Bregman method as described in Herrmann et al. (2015). The Julia code follows largely the algorithm

in Listing 6, with an additional variable and sparsity promotion through soft thresholding.

[Figure 5 about here.]

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 (Williams et al., 2009; Andreolli et al., 2014; Louboutin et al., 2017a), 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 Louboutin et al. (2018a) and Luporini et al. (2018), 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 differ-

ent 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.

CONCLUSIONS

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.

ACKNOWLEDGMENTS

We would like to thank Henryk Modzelewski (The University of British Columbia) for the technical support and the Julia Operator Library (JOLI). Furthermore, we would like to acknowledge Charles Jones (Osokey Ltd.) for the useful discussions on early versions of the software and the manuscript, as well as Keegan Lensink (The University of British Columbia) for SeisIO. This research was carried out as part of the SINBAD project with the support of the member organizations of the SINBAD Consortium. Finally, we would like to acknowledge the University of British Columbia (UBC), where part of this research was carried out.

APPENDIX A

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 (Louboutin et al., 2017b, 2018b). 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/v0.6/JUDI/src/Python/JAcoustic_codegen.py`.

APPENDIX B

Verification framework

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 (Terentyev, 2009; Symes and Dong, 2010). Figure 6 shows trace comparisons of JUDI and iWave for two different velocity models and validates that both codes 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.

[Figure 6 about here.]

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 (Claerbout, 1992). 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 1).

```
# 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 8: 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 7) behave as predicted by Taylor's theorem

for $h \to 0$:

$$\begin{aligned} &\Phi(\mathbf{m}_0 + h \cdot \delta\mathbf{m}) - \Phi(\mathbf{m}_0) = \mathcal{O}(h) \\ &\Phi(\mathbf{m}_0 + h \cdot \delta\mathbf{m}) - \Phi(\mathbf{m}_0) - h \cdot \nabla\Phi(\mathbf{m}_0)^\top \delta\mathbf{m} = \mathcal{O}(h^2). \end{aligned} \tag{3}$$

[Figure 7 about here.]

REFERENCES

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. 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, 2016, TensorFlow: A system for large-scale machine learning: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 265–283.

Andreolli, C., P. Thierry, L. Borges, C. Yount, and G. Skinner, 2014, Genetic algorithm based auto-tuning of seismic applications on multi and manycore computers: Presented at the EAGE Workshop on High Performance Computing for Upstream.

Balay, S., 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, 2016, PETSc users manual. Argonne National Laboratory, 3.7 ed.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah, 2017, Julia: A fresh approach to numerical computing: Society for Industrial and Applied Mathematics (SIAM) Review, **59**, 65–98.

Bezanson, J., S. Karpinski, V. B. Shah, and A. Edelman, 2012, Julia: A fast dynamic language for technical computing: Computing Research Repository.

Bottou, L., 2010, *in* Large-Scale Machine Learning with Stochastic Gradient Descent: Physica-Verlag HD, 177–186.

Cervantes, A. M., A. Wächter, R. H. Tütüncü, and L. T. Biegler, 2000, A reduced space interior point strategy for optimization of differential algebraic systems: Computers and Chemical Engineering, **24**, 39 – 51.

Claerbout, J., 1992, Earth soundings analysis: Processing versus inversion: Blackwell Sci-

entific Publications.

Da Silva, C., and F. J. Herrmann, 2017, A unified 2D/3D large scale software environment for nonlinear inverse problems: Computing Research Repository.

Dagum, L., and R. Menon, 1998, OpenMP: An industry-standard API for shared-memory programming: Institute of Electrical and Electronics Engineers (IEEE) Computational Science and Engineering, **5**, 46–55.

Dong, S., J. Cai, M. Guo, S. Suh, Z. Zhang, B. Wang, and Z. Li, 2012, Least-squares reverse time migration: Towards true amplitude imaging and improving the resolution: 82nd Annual International Meeting, SEG, Expanded Abstracts, **79**, 1–5.

Fomel, S., P. Sava, I. Vlad, L. Yang, and V. Bashkardin, 2013, Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments: Journal of Open Research Software, **1**.

Griewank, A., and A. Walther, 2000, 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, **26**, 19–45.

Guitton, A., and W. W. Symes, 2003, Robust inversion of seismic data using the Huber norm: GEOPHYSICS, **68**, 1310–1319.

Hassanzadeh, S., and C. C. Mosher, 1997, Javaseis: Web delivery of seismic processing services: 2055–2057.

Heroux, M. A., 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, 2005, An overview of the Trilinos project: Association for Computing Machinery (ACM) Transactions on Mathematical Software, **31**, 397–423.

Herrmann, F. J., P. Moghaddam, and C. Stolk, 2008, Sparsity- and continuity-promoting seismic image recovery with curvelet frames: Applied and Computational Harmonic Analysis, **24**, 150–173.

Herrmann, F. J., N. Tu, and E. Esser, 2015, Fast "Online" Migration with Compressive Sensing: 77th EAGE Conference and Exhibition.

Huang, G., R. Nammour, and W. Symes, 2017, Full-waveform inversion via source-receiver extension: GEOPHYSICS, **82**, R153–R171.

Johnson, S., 2017, Calling Python functions from the Julia language: `https://github.com/JuliaPy/PyCall.jl`, (accessed July 27, 2018).

Koehn, D., 2017, Sava: `https://github.com/daniel-koehn/SAVA`, (accessed May 21, 2018).

Krischer, L., A. Fichtner, S. Zukauskaite, and H. Igel, 2015, Large-scale seismic inversion framework: Seismological Research Letters, **86**, 1198.

Kukreja, N., J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, 2018, High-level Python abstractions for optimal checkpointing in inversion problems: ArXiv e-prints.

Kukreja, N., M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman, 2016, Devito: Automated fast finite difference computation: Computing Research Repository.

Lange, M., N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, 2016, Devito: Towards a generic finite difference DSL using symbolic Python: Computing Research Repository.

Lensink, K., 2017, SeisIO.jl: `https://github.com/slimgroup/SeisIO.jl`, (accessed February 11, 2018.

Logg, A., K.-A. Mardal, and G. Wells, 2012, Automated solution of differential equations

by the finite element method: Springer, volume **84** *of* Lecture Notes in Computational Science and Engineering.

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

Louboutin, M., M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, 2018a, Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration: ArXiv preprints.

Louboutin, M., P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, 2017b, Full-waveform inversion, Part 1: Forward modeling: The Leading Edge, **36**, 1033–1036.

———, 2018b, Full-waveform inversion, Part 2: Adjoint modeling: The Leading Edge, **37**, 69–72.

Luporini, F., D. A. Ham, and P. H. J. Kelly, 2016, An algorithm for the optimization of finite element integration loops: Computing Research Repository.

Luporini, F., M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, 2018, Architecture and performance of Devito, a system for automated stencil computation: ArXiv preprints.

Modzelewski, H., 2017, JOLI - Julia Operator Library: `https://github.com/slimgroup/JOLI.jl`, (accessed October 26, 2017).

Nemeth, T., C. Wu, and G. T. Schuster, 1999, Least-squares migration of incomplete reflection data: GEOPHYSICS, **64**, 208–221.

Nickolls, J., I. Buck, M. Garland, and K. Skadron, 2008, Scalable parallel programming with CUDA: Queue, **6**, 40–53.

Padula, A. D., S. D. Scott, and W. W. Symes, 2009, A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms: Association for Computing Machinery (ACM) Transactions on Mathematical Software, **36**, 8:1–8:36.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, 2017, Automatic differentiation in PyTorch: Presented at the Conference on Neural Information Processing Systems (NIPS).

Plessix, R.-E., 2006, A review of the adjoint-state method for computing the gradient of a functional with geophysical applications: Geophysical Journal International, **167**, 495.

Rathgeber, F., D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G. Bercea, G. R. Markall, and P. H. J. Kelly, 2015, Firedrake: Automating the finite element method by composing abstractions: Computing Research Repository.

Ruthotto, L., E. Treister, and E. Haber, 2016, jinv - a flexible Julia package for PDE parameter estimation: Computing Research Repository.

Schmidt, M., E. van den Berg, M. Friedlander, and K. Murphy, 2009, Optimizing costly functions with simple constraints: A limited-memory projected Quasi-Newton algorithm: Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS) 2009, 456–463.

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

Symes, W. W., 2007, Reverse time migration with optimal checkpointing: GEOPHYSICS, **72**, SM213–SM221.

———, 2017, The search for a cycle-skipping cure: An overview: Presented at the Institute for Pure and Applied Mathematics (IPAM): Computational Issues in Oil Field Applica-

tions.

Symes, W. W., and S. Dong, 2010, The IWAVE++ inversion framework: `http://trip.rice.edu/reports/2010/dong2.pdf`, (accessed March 5, 2018).

Symes, W. W., D. Sun, and M. Enriquez, 2011, From modelling to inversion: Designing a well-adapted simulator: Geophysical Prospecting, **59**, 814–833.

Tan, C., S. Ma, Y.-H. Dai, and Y. Qian, 2016, Barzilai-Borwein Step Size for Stochastic Gradient Descent: ArXiv e-prints.

Tang, Y., and B. Biondi, 2009, Least-squares migration/inversion of blended data: 79th Annual International Meeting, SEG, Expanded Abstracts.

Tarantola, A., 1984, Inversion of seismic reflection data in the acoustic approximation: GEOPHYSICS, **49**, 1259–1266.

Terentyev, I., 2009, A software framework for finite difference simulation: Technical report, Tech. Rep. 09-07, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA.

Thorbecke, J., 2017, Opensource: `https://github.com/JanThorbecke/OpenSource`, (accessed June 18, 2018).

Tompson, J., and K. Schlachter, 2012, Introduction to the OpenCL programming model.

van den Berg, E., and M. P. Friedlander, 2013, Spot - a linear operator toolbox: `https://github.com/mpf/spot`, (accessed July 20, 2018).

van den Berg, E., M. P. Friedlander, G. Hennenfent, F. J. Herrmann, R. Saab, and O. . Yilmaz, 2009, Algorithm 890: Sparco: A testing framework for sparse reconstruction: Association for Computing Machinery (ACM) Transactions on Mathematical Software, **35**, 1–16.

van Leeuwen, T., A. Y. Aravkin, H. Calandra, and F. J. Herrmann, 2013, In which domain

should we measure the misfit for robust full waveform inversion?: Presented at the EAGE Annual Conference Proceedings.

van Leeuwen, T., and F. J. Herrmann, 2013, Mitigating local minima in full-waveform inversion by expanding the search space: Geophysical Journal International, **195**, 661–667.

Virieux, J., and S. Operto, 2009, An overview of full-waveform inversion in exploration geophysics: GEOPHYSICS, **74**, WCC127–WCC152.

Williams, S., A. Waterman, and D. Patterson, 2009, Roofline: An insightful visual performance model for multicore architectures: Communications of the Association for Computing Machinery (ACM), **52**, 65–76.

Witte, P. A., M. Louboutin, and F. J. Herrmann, 2017, The Julia Devito inversion framework (JUDI): `https://github.com/slimgroup/JUDI.jl`, (accessed August 31, 2018).

Witte, P. A., M. Louboutin, K. Lensink, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, 2018, Full-waveform inversion, Part 3: Optimization: The Leading Edge, **37**.

Zeng, G. L., and G. T. Gullberg, 2000, Unmatched projector/backprojector pairs in an iterative reconstruction algorithm: Institute of Electrical and Electronics Engineers (IEEE) Transactions on Medical Imaging, **19**, 548–555.

Zhang, S., A. E. Choromanska, and Y. LeCun, 2015, Deep learning with elastic averaging SGD: Advances in Neural Information Processing Systems, 685–693.

LIST OF FIGURES

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. 52

2 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). 53

3 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. 54

4 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). 55

5 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. 56

6 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. 57

7 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 $h \rightarrow 0$, which verifies that the gradient is implemented correctly. . 58

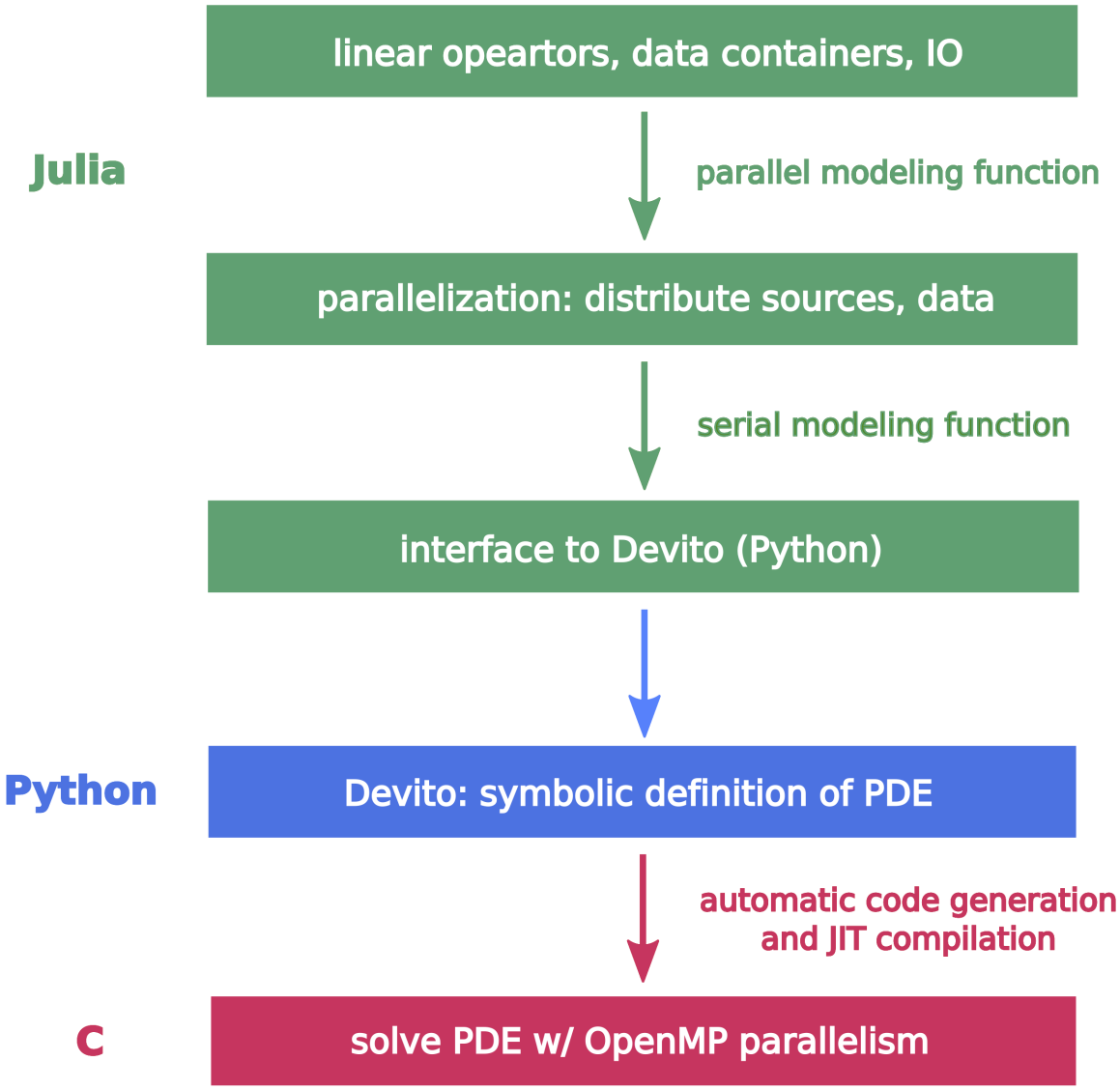

Figure 1: Software hierarchy of JUDI and its interface to the wave equation solver Devito. The uppermost software layer contains matrix-free operators that allow expressing PDE solvers and sampling operators as linear algebra operations. For solving multiple PDEs, data is first distributed to the available 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.

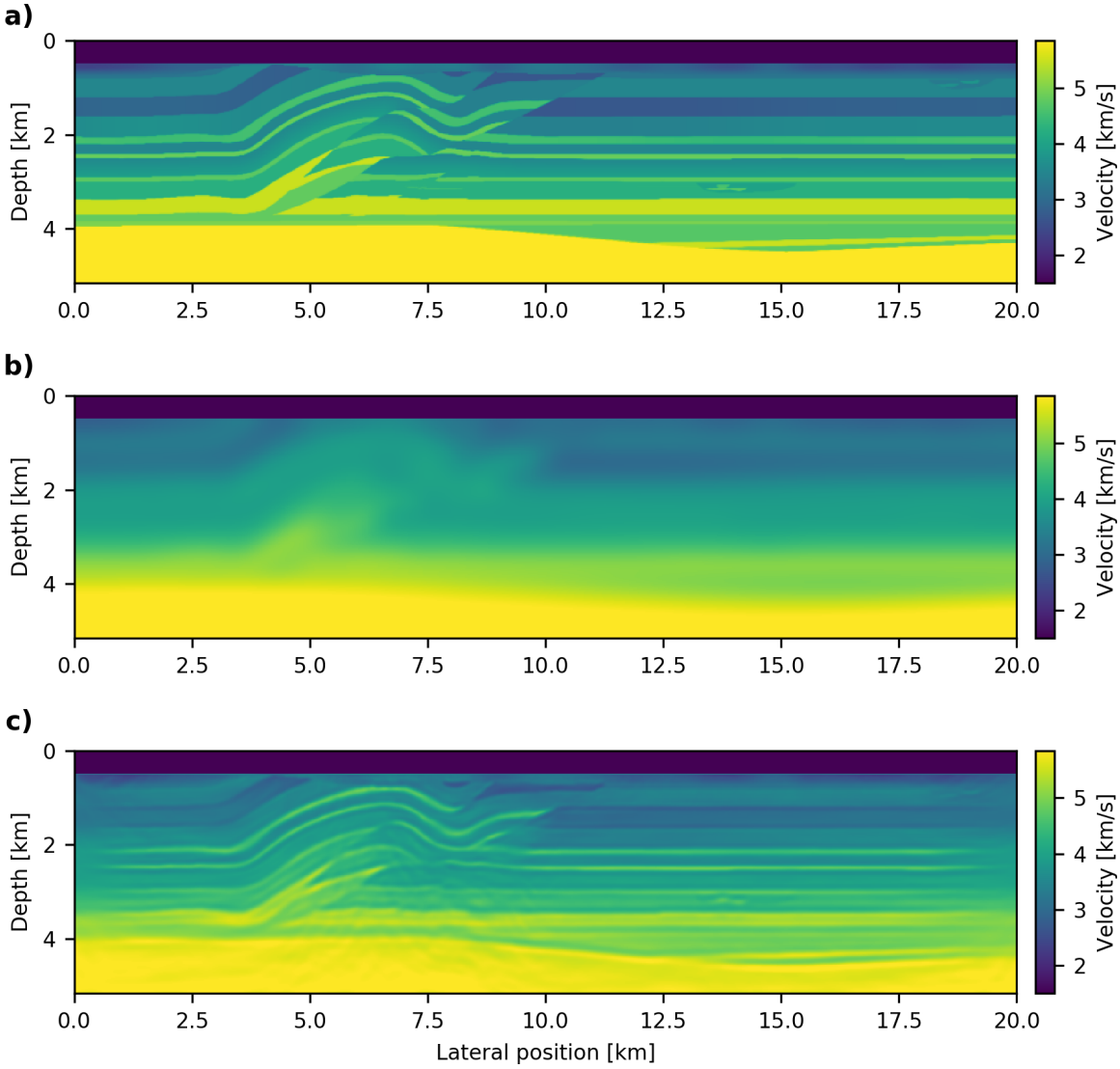

Figure 2: Overthrust velocity model for our 2D FWI case study (a), initial model (b) and recovered model after 20 iterations of stochastic gradient descent with bound constraints and a backtracking line search (c).

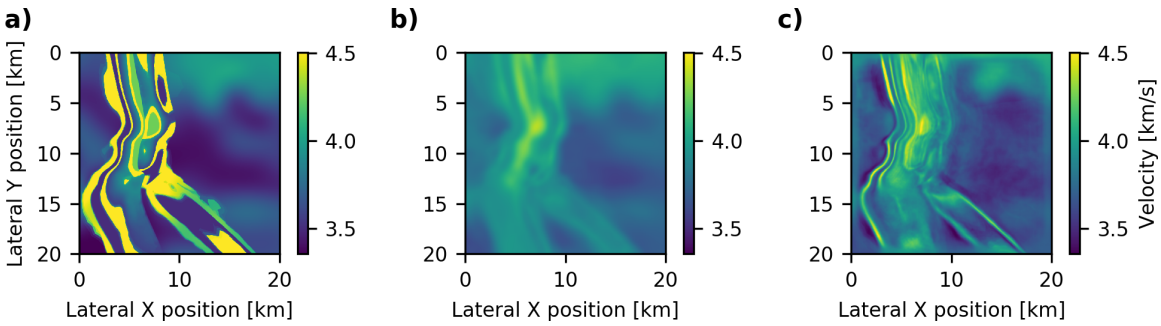

Figure 3: Depth slice through the original 3D Overthrust model (a), the initial model (b) and the recovered model after 15 function evaluations with minConf's spectral projected gradient algorithm (c). Some parts of the recovered model are cycle skipped, but overall minConf's SPG algorithm was able to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

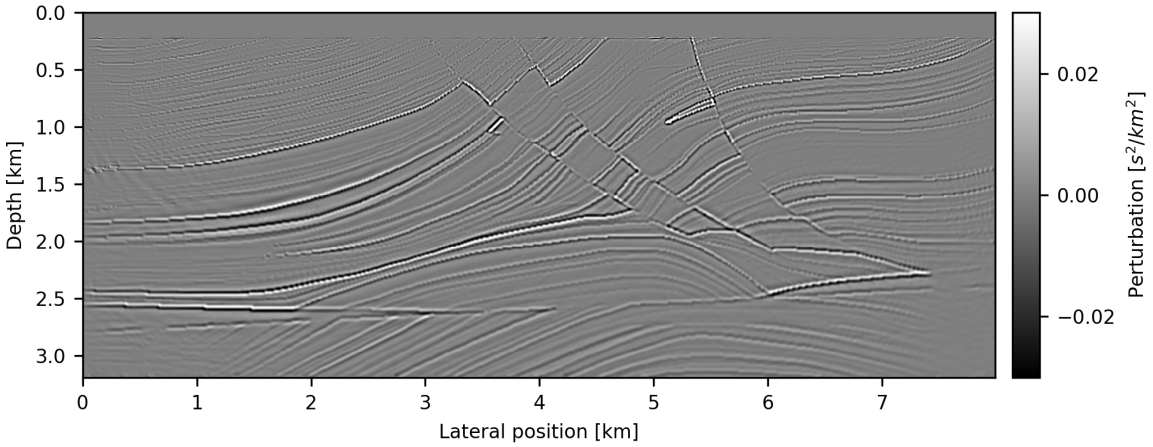

Figure 4: LS-RTM image of the Marmousi model after 20 iterations of the elastic average SGD algorithm. In each iteration, the 10 workers calculate their new image from single randomly selected shot and the master updates the central variable (shown here after the final iteration).

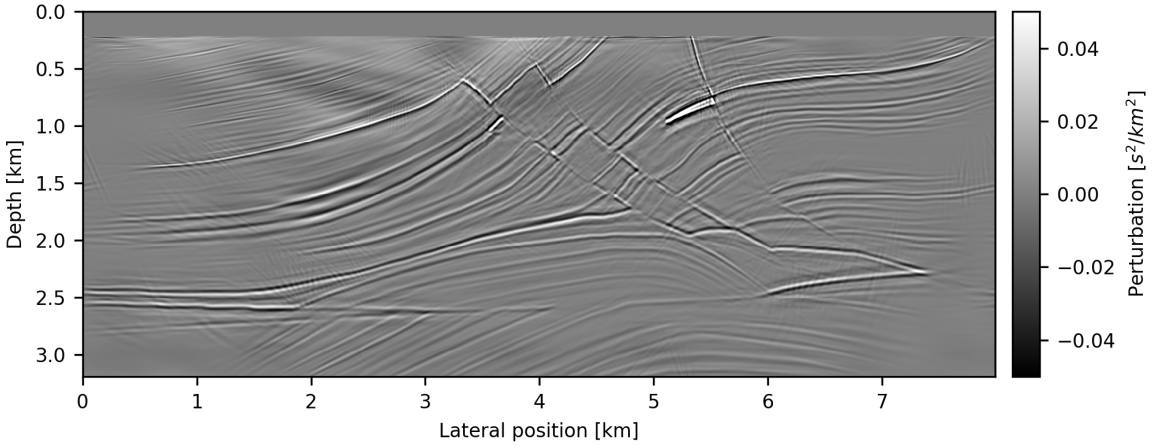

Figure 5: Imaging result after 32 iterations of sparsity-promoting LS-RTM with on-the-fly Fourier transforms. By only saving a few frequency-domain wavefields, this method only requires a fraction of the memory of conventional time-domain LS-RTM and therefore scales to large-scale models.

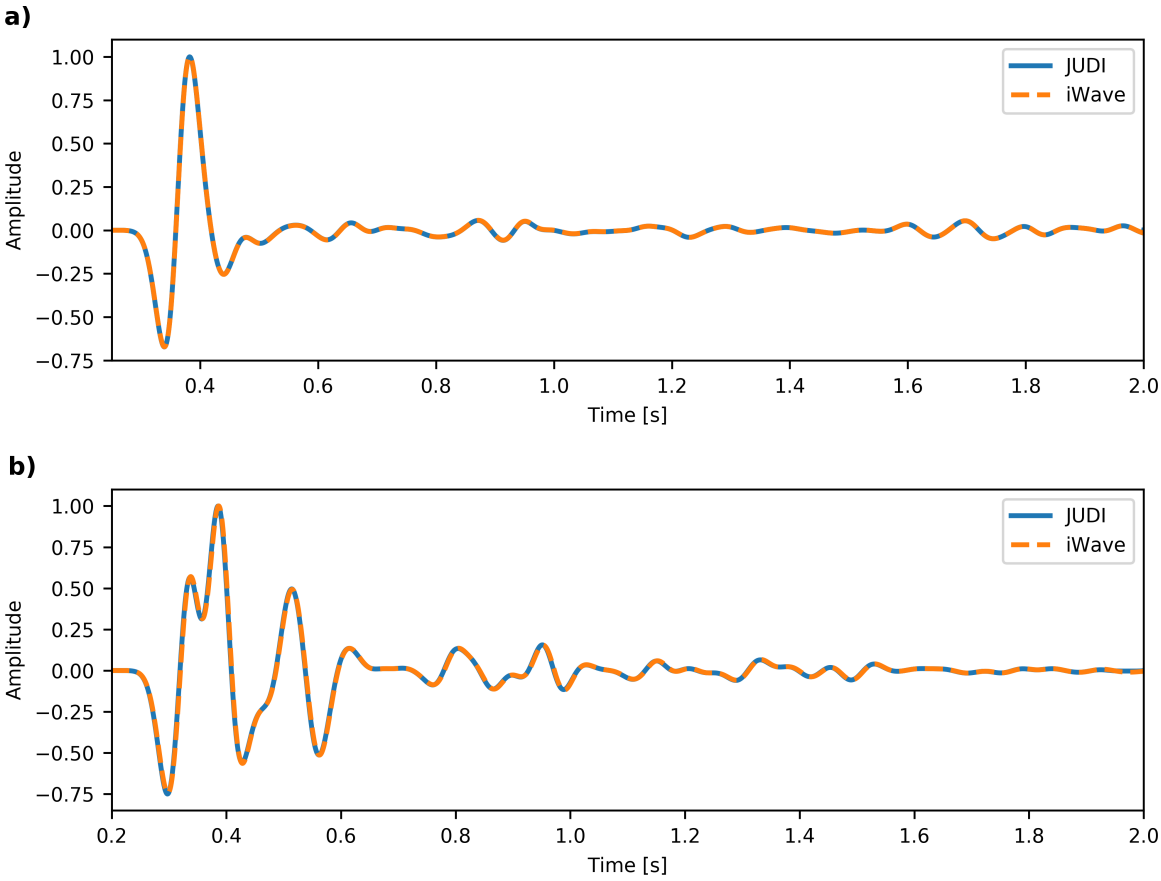

Figure 6: Comparison of a single traces from seismic shot records that were modeled with JUDI and iwave. Figure (a) was generated using the 2D Marmousi model and Figure (b) was modeled with the 2D Overthrust model.

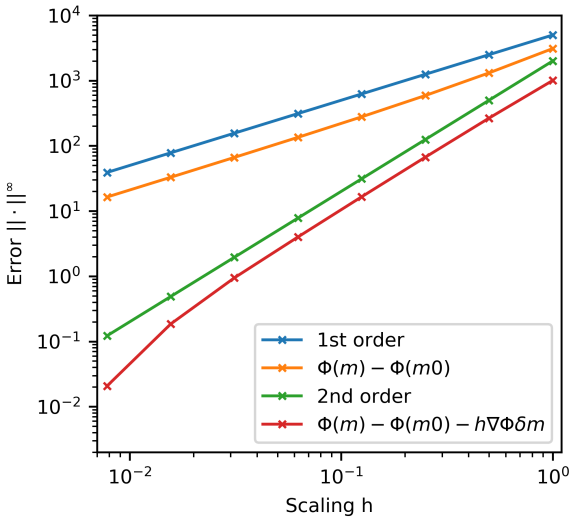

Figure 7: Taylor error test for the implementation of the FWI objective function and gradient. Using the gradient information causes the error to decay with 1st order as $h \to 0$, which verifies that the gradient is implemented correctly.