A Large-Scale Time-Domain Seismic Modeling and Inversion Workflow in Julia

Philipp A. Witte*[1], Mathias Louboutin[1], Gerard Gorman[2] and Felix J. Herrmann[1]
[1]Seismic Lab. for Imaging and Modeling, University of British Columbia, Canada
[2] Department of Earth Science and Engineering, Imperial College London, UK

***Abstract*—We present our initial steps towards the development of a large-scale seismic modeling workflow in Julia that provides a framework for wave equation based inversion methods like full waveform inversion or least squares migration. Our framework is based on the Devito, a finite difference domain specific language compiler that generates highly optimized and parallel code. We develop a flexible workflow that is based on abstract matrix-free linear operators and enables developers to write code that closely resembles the underlying math, while at the same time leveraging highly optimized wave equation solvers, allowing us to solve large-scale three-dimensional inverse problems.**

I. Introduction

Software packages for seismic modeling and inversion frameworks often suffer from a trade-off between performance and maintainability. Many packages written in lower-level languages such as Fortran or C, even though being computationally efficient and able to scale to large problem sizes, are tailored to a specific application, which makes it difficult for researchers to adapt new ideas and algorithms. Furthermore, these codes are difficult to maintain, debug and test, making them often erroneous and inaccurate. Higher-level script-based languages as Python, Matlab or Perl are on the other hand generally slower and often do not scale on high-performance computing (HPC) environments.

In this work, we try to overcome this issue by building our code in a hierarchical and modular fashion, in which we separate the wave equations solves from high-level abstractions for algorithm design. The concept of abstract linear operators is adapted from earlier works of [1] and [2] and has been applied to frequency domain modeling and full waveform inversion (FWI) workflows by [3] and [4]. In this work, we build a framework based on these concepts for time-domain seismic modeling and inversion and we use a domain specific language (DSL) for fast and efficient wave equations solves.

Our framework is written in Julia, a relatively new dynamic programming language particularly designed for numerical computing, which combines strengths of both statically and dynamically typed languages, with features like optional typing, just-in-time compilation (JIT) and function overloading [5]. Our code consists of basically three independent layers that can be individually modified and extended: high-level abstractions (e.g. linear operators, objective functions), parallelization and data distribution and wave equation solves. For solving 2 and 3D wave equations, we use the Devito toolbox, which autogenerates C code with optimized finite difference stencils from symbolic Python expressions, that can be called directly from Julia [6]. Overall this provides a framework that is on the one hand readable and easy to use, but on the other hand still scales to large, industry-size problems.

II. Code design

A. Code hierarchy

Our Julia software framework is designed in a hierarchical fashion with independent building blocks, that can be independently interchanged or modified without affecting the other layers (Figure 1). This makes the software flexible and easy to maintain. The topmost layer of our software is designed to handle two general types of inverse problems: linear problems, like least squares migration and nonlinear problems, like FWI. Our goal is to design the user interface in such a way, that a variety of existing black box optimization libraries and linear solvers can be used to tackle these problems.

FWI is a nonlinear inverse problem, because the objective function is nonlinear with respect to the parameter $\mathbf{m}$ that we invert for (seismic velocity or squared slowness):

$$\underset{\mathbf{m}}{\text{minimize}} \quad \Phi(\mathbf{m}) = \sum_{i=1}^{N} \frac{1}{2} ||\mathbf{P}_i \mathbf{A}(\mathbf{m})^{-1} \mathbf{q}_i - \mathbf{d}_i||_2^2. \tag{1}$$

The index i corresponds to the ith source location of a seismic experiment, $\mathbf{P}_i$ is the sampling operator that restricts the wavefields to the receiver location, $\mathbf{A}(\mathbf{m})$ is the discretized 2/3D acoustic wave equation, $\mathbf{q}_i$ is the source wavelet function and $\mathbf{d}_i$ is the ith observed seismic shot record.

Nonlinear optimization problems like this one can be solved using gradient- and/or Hessian based optimization algorithms, such as the nonlinear Conjugate Gradient Method (CG) or Newton-type methods like L-BFGS [8]. Software libraries with implementations of these algorithms generally require the same input, namely a function or a pointer to a function that returns the gradient and function value of the objective function for the current model iterate. For the convergence of these methods, it is crucial, that the gradients of the objective are implemented correctly. We implement such a function for the FWI objective function, which allows us to use many different black box optimization libraries for the inversion.

The second class of problems we deal with, are linear problems, i.e. the objective function is linear with respect to

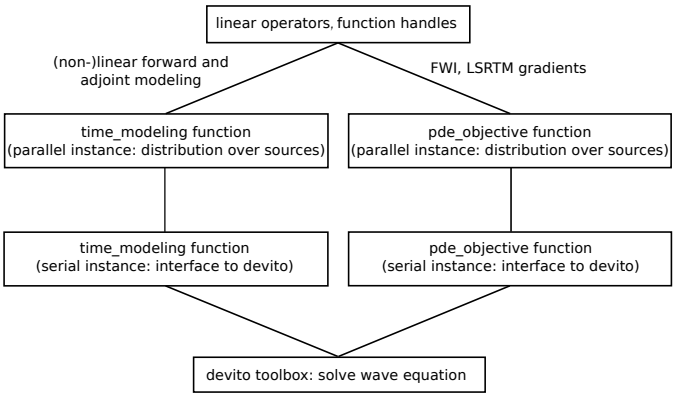

Fig. 1. Code hierarchy of the Julia time-domain modeling and inversion framework

the optimization variables. An example for these types of problems is least squares migration:

$$\underset{\delta\mathbf{m}}{\text{minimize}} \quad \Psi(\delta\mathbf{m}) = \frac{1}{2}||\mathbf{J}\delta\mathbf{m} - \delta\mathbf{d}||_2^2, \tag{2}$$

where $\mathbf{J}$ is the linearized Born modeling operator defined as $\mathbf{J} = \mathbf{A}(\mathbf{m})^{-1}\text{diag}\Big(\frac{\partial\mathbf{A}}{\partial\mathbf{m}}\mathbf{A}(\mathbf{m})^{-1}\mathbf{q}\Big)$, $\delta\mathbf{m}$ is the model perturbation (seismic image), and $\delta\mathbf{d}$ is the observed seismic reflection data.

Problems like this one can be solved with linear iterative solvers such as LSQR, the linearized Bregman Method or the $\ell 1$ Spectral Projected Gradient Method (SPGℓ1) [9] [10]. Numerical implementations of these algorithms generally take the explicit matrix $\mathbf{J}$ as well as the observed data $\delta\mathbf{d}$ as an input. In our case, the matrix $\mathbf{J}$ is the time stepping matrix for linearized modeling of the full seismic experiment and due to its size can never be formed explicitly. Therefore, we build a matrix-free linear operator, based on the Julia Linear Operator Package [11], that can be used in the same way as an explicit matrix, i.e. it can be transposed, applied to a matrix or vector and passed to black box linear solvers. We also implement the nonlinear forward modeling operator $\mathbf{F} = \mathbf{A}(\mathbf{m})^{-1}$, which is nonlinear with respect to $\mathbf{m}$, but can be applied as a linear operator to a seismic source: $\mathbf{d} = \mathbf{F} \cdot \mathbf{q}$.

The linear operators and objective functions are simply wrappers for the `time_modeling`() and the `pde_objective`() function, which form the intermediate layer of our software framework. These functions form the interface to the Devito toolbox and set up the necessary parameters that are required. After the computations are completed, these functions gather the results that are requested from the top layer, e.g. wavefields, gradients or shot records.

Devito solves one wave equation for a specified source location at a time. We use Julia's function overloading, called $multiple\ dispatch$, to implement a serial and a parallel instance of the `time_modeling` and `pde_objective` function. If the functions are called for more than one source, the parallel instance of the functions distributes the sources over the available nodes and workers and then recursively calls its serial instance to interface Devito. This way, the task parallelism is completely separated from the rest of the code and does not affect the readability of the other building blocks.

B. Parallelization

Julia uses a message passing based parallel framework, that uses one-sided communication and requires only managing of the master process [5]. For our framework, we parallelize in Julia over sources of the seismic experiment, where the workers of the parallel pool can have either shared or distributed memory. The Devito toolbox itself uses OpenMP to parallelize over the computational domain (velocity model).

To avoid unbalanced workloads we use dynamic scheduling to assign sources to the workers through a Julia feature called $channels$. A channel is a shared queue that can hold abstract objects and multiple workers can read from and write to a channel. In our case, the channel has a length of N, where N is the number of seismic experiments or source positions. The channel is filled with the source/receiver positions for each experiment and the workers from the parallel pool dynamically take out source positions from the channel as long as the channel is open, i.e. has unprocessed source positions. These function calls are non-blocking, i.e. the main process sends out all computations and returns immediately. Once the computations are completed, the results (shot records, gradients) are fetched from the remote workers.

Furthermore Julia allows us to easily build in resilience to hardware failures. Julia has a built-in function called `retry(f, n, delay)` that takes as an input a function `f`, that is retried up to `n` times in case of an exception, for example if one of the workers is interrupted or crashes [5]. We wrap the parallel instances of our `time_modeling` and `pde_objective` function in the `retry` function and test the resilience by killing a worker from the terminal during a parallel modeling job. After crashing a worker, Julia resends the work load from the terminated worker to one of the remaining ones and continues to run the program. The built-in resilience allows to run jobs on large HPC clusters, where hardware failures are fairly common.

C. Devito interface

Devito is a finite difference domain specific language (DSL), that generates optimized stencils from high-level symbolic Python equations and performs automatic code generation and just-in-time (JIT) compilation [6]. Devito is designed to generate stencil-based C code that is not only fast, but that optimally utilizes the hardware for a given numerical intensity, as described by the roofline model [12]. In the original Devito software, setting up the symbolic PDE, generating the optimized stencil and C code as well as calling the C code itself is all done in Python. For our framework, we only need to interface Python to set up the PDE and perform the C code generation and we then call the generated C code directly from Julia. We use the Julia PyCall library to interface Devito and to pass the information to Python that is necessary to set up the PDE, i.e. grid information, source and receiver locations, wavelet et cetera [13]. The C code that is generated from

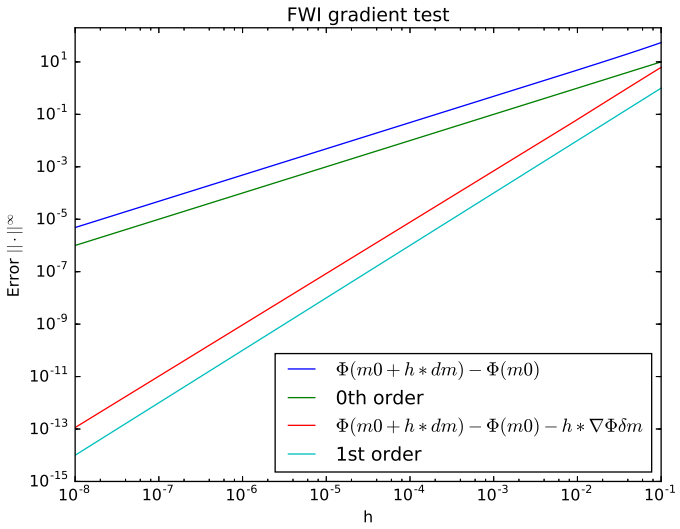

Fig. 2. Taylor error of the FWI objective function for $0th$ and $1st$ order.

Devito can then be called directly from Julia, since Julia allows direct C and Fortran function calls [5]. We want to point out that no variables and data copies are required, since all input and return arguments are passed by reference, which allows to scale our code to very large problem sizes.

III. Unit testing

As mentioned earlier, having a correct implementation of the gradient is crucial for the convergence of the optimization algorithms. To ensure that the gradient for the FWI objective function is implemented correctly, we test the behaviour of the $0th$ and $1st$ order Taylor errors. Assuming that $\Phi(\mathbf{m})$ is a smooth function, we ensure that for a reference model $\mathbf{m}_0$ and a model perturbation $h \cdot \delta\mathbf{m}$, the Taylor errors behave as predicted by Taylor's theorem as $h \rightarrow 0$:

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

The Taylor error is plotted in logarithmic scale against h and shows that in fact, the gradients in our software framework are implemented correctly (Figure 2).
For the convergence of linear solvers, it is important to ensure that our linear operators have a correct implementation of the action of the adjoint operator, i.e. that

$$\langle \mathbf{b}, \mathbf{A}\mathbf{x} \rangle = \langle \mathbf{A}^T\mathbf{b}, \mathbf{x} \rangle \tag{4}$$

holds within machine precision for a matrix-free linear operator $\mathbf{A}$ and two random vectors $\mathbf{x}$ and $\mathbf{b}$. We perform the adjoint test for both linear operators $\mathbf{F}$ and $\mathbf{J}$ with vectors that are in the range of the wave equation to prevent violation of the CFL and stability conditions (Table I).

IV. Applications

The high-level abstractions of our Julia software framework and the separation from the low level computations, make it easy to implement algorithms and mathematical concepts. For example, rather than using a black box linear solver for the least squares migration problem (Equation 2), it is possible to implement a custom solver in very few lines, with the code closely following the mathematical formulation. For example, we can use our linear operators to implement a sparsity-promoting least squares migration algorithm using the linearized Bregman method [9] [14]:

TABLE I
Adjoint test for matrix-free linear operators

Operator	$\mathbf{b}^T\mathbf{A}\mathbf{x}$	$\mathbf{x}^T\mathbf{A}^T\mathbf{b}$	Relative Difference
$\mathbf{F}$	-71.126979	-71.126980	-1.628620e-8
$\mathbf{J}$	0.203647	0.203647	8.471093e-7

```
# Set up matrix free linear operator
J = opJ(model,params,srcNum,wavelet)
x=0; z=0
for n = 1:numIterations
    # Calculate residual
    r = J*x - d
    # Calculate gradient
    g = J'*r
    # Calculate step size
    t = (norm(r)/norm(g))^2
    # Update variables
    z = z - t*g
    x = softThreshold(z)
end
```

For nonlinear optimization problems like FWI (Equation 1), we can use the implementation of the objective function and pass it to a generic black box optimization toolbox. We have the possibility to choose from a large variety of libraries and algorithms. In the following example, we use the NLopt library from [15] and perform FWI using L-BFGS with bound constrains. Each library may request slightly different function pointers, so we set up a custom function for the NLopt library, which requires also the current gradient as a function argument and then overwrites it with the updated gradient:

```
# Custom function pointer for NLopt
count=0
function NLoptObjective!(x,grad)
    f,g[1:end] = pdeObjective(model, ...)
    global count
    count += 1
    return fval
end

# L-BFGS with bound constraints
opt = Opt(:LSLBFGS, prod(n))
lowerBounds!(opt, mMin)
upperBounds!(opt, mMax)
minObjective!(opt, NLoptObjective!)
maxeval!(opt, 20)
(minf, minx, ret) = optimize(opt, m0)
```

As mentioned before, the code that is generated from Devito for the wave equation solves is optimized in terms of achieving optimal hardware usage for a given arithmetic intensity [12]. However, to provide an idea of the absolute speed of the code, we have assembled some timings for forward and adjoint wave equations solves for a given model size (Table II). Obviously, these numbers will vary depending on the hardware and is only intended to provide a general idea of the code speed.

For a model size of 200^3 grid points (with 1 second recording time, sampled at 4 ms and using a 128GB node) we are able to save all wavefields for the gradient calculation in RAM. However for calculating a gradient on a domain with 580^3 grid points, we need to write all wavefields to disk, which makes the calculation considerably slower. This problem can be addressed by subsampling the wavefields or checkpointing.

V. Conclusions

We developed a software framework for time-domain seismic modeling an inversion that is based on high-level abstractions for solving linear and nonlinear inverse problems, while at the same time relying on a super fast and efficient wave equations solver. Our framework is build upon a hierarchical code design that separates high-level abstractions for algorithm design and optimization from the low-level wave equation solves and parallelization, therefore yielding a code base that is easy to maintain, test and modify. New mathematical concepts can be easily adapted using abstract linear operators or objective functions. For solving forward and adjoint wave equations, we interface the Devito tool, which autogenerates optimized finite difference kernels from symbolic Python PDEs and scales to arbitrary problem sizes, allowing us to efficiently solve wave equations on large three-dimensional domains.

Our Julia framework is furthermore resilient to hardware failures and reassigns workloads from terminated nodes or workers without crashing the entire program. This prevents us from having to save many checkpoints and allows us to use our software efficiently on large HPC clusters and cloud environments.

TABLE II
Run time for forward modeling and gradient calculation of a single source on 3D velocity model for different numbers of grid points n and recording time t. Results were computed on a single node with 128 GB RAM and 20 Intel Xeon Processors E5-2690 v2. For 200^3 grid points, all computations are in-core, whereas for 580^3 grid points, the calculations for the gradient are done out-of-core.

Operation	Run time (n = 200^3, t = 1 [s])	Run time (n = 580^3, t = 4 [s])
forward modeling: $\mathbf{F} \cdot \mathbf{q}$	139 seconds	41 minutes
Born modeling: $\mathbf{J} \cdot \delta\mathbf{m}$	162 seconds	71 minutes
FWI gradient: $\mathbf{J}^T \cdot \delta\mathbf{d}$	239 seconds	20 hours

References

[1] W. Symes, D. Sun and M. Enriquez, *From modelling to inversion: designing a well-adapted simulator*, Geophysical Prospecting, EAGE, August 2011

[2] R.A. Bartlett, B.G. Van Bloemen Waanders and M.A. Heroux, *Vector Reduction/Transformation Operators*, ACM Transactions on Mathematical Software, Vol V, No. N, August 2003, Pages 1–25

[3] T. Van Leeuwen, *A parallel matrix-free framework for frequency-domain seismic modelling, imaging and inversion in Matlab*, Internal report, https://www.slim.eos.ubc.ca/Publications/Public/TechReport/2012/vanleeuwen2012smii/vanleeuwen2012smii.pdf, July 2012

[4] C. Da Silva and F. J. Herrmann, *A unified 2D/3D software environment for large scale time-harmonic full waveform inversion*, SEG Technical Program Expanded Abstracts, 2016

[5] N/A, *Julia Documentation*, http://docs.julialang.org/en/release-0.5/

[6] 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*. Accepted for PyHPC2016, to appear in ACM SIGHPC, 2016, arxiv:1609.03361

[7] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange and G. Gorman, *Devito: Automated fast finite difference computation.*, The International Conference for High Performance Computing, Networking, Storage and Analysis; Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, 2016

[8] J. Nocedal and S.J. Wright, *Numerical Optimization*, Second Edition, Springer, 2006

[9] W. Yin, *Analysis and Generalizations of the Linearized Bregman Method*, SIAM Journal on Imaging Sciences, 3, 4, pp. 856–877, 2010

[10] E. van den Berg and M.P. Friedlander, *Probing the Pareto frontier for basis pursuit solutions*, SIAM Journal on Scientific Computing, Vol. 31, No. 2, pp. 890-912, 2008

[11] A. Siqueira, *A Julia Linear Operator Package*, Github Repository, https://github.com/JuliaSmoothOptimizers/LinearOperators.jl

[12] M. Louboutin, M. Lange, N. Kukreja, F. Herrmann, and G. Gorman, *Performance prediction of finite-difference solvers for different computer architectures*, submitted to Computers & Geosciences, 2016

[13] S. G. Johnson, *Calling Python functions from the Julia language*, Github Repository, https://github.com/JuliaPy/PyCall.jl

[14] F. J. Herrmann, N. Tu and E. Esser, *Fast "online" migration with Compressive Sensing*, EAGE abstract, 2015

[15] S. G. Johnson, *The NLopt module for Julia*, Github Repository, https://github.com/JuliaOpt/NLopt.jl