REPORT
doi:10.26443/seismica.v4i2.1475

A brief exploration of open-source gradient-based numerical optimization Python libraries for full-waveform inversion

Oscar Mojica * [1,2]

[1]Supercomputing Center for Industrial Innovation, SENAI CIMATEC, Salvador, Brazil, [2]National Institute of Science and Technology of Petroleum Geophysics (INCT-GP/CNPQ), Centro de Pesquisa em Geofísica e Geologia (CPGG-UFBA) Instituto de Geociências, sala 206-E, Salvador, Brazil

Author contributions: *Conceptualization*: Oscar Mojica. *Methodology*: Oscar Mojica. *Validation*: Oscar Mojica. *Formal Analysis*: Oscar Mojica. *Writing - Original Draft*: Oscar Mojica. *Writing - Review & Editing*: Oscar Mojica.

Abstract Geoscientists favor Python for its user-friendly interface and scientific packages that support application implementation. Python's capabilities make it particularly useful for seismic full waveform inversion (FWI), which can see its implementation time reduced by making use of its extensive library collection. We compare four open-source gradient-based optimization Python packages—`scipy.optimize`, sotb-wrapper, NLopt, and PyROL—for solving the FWI optimization problem. The comparison is based on the packages' core features, documentation, and learning curves evaluated through the implementation of a 2D time-domain FWI application, built using the Devito modeling engine along with the aforementioned optimization packages. We detail how one can use a particular solver from each package for the solution of a bound-constrained optimization problem such as FWI. The open-source FWI template models used to obtain the numerical results are provided.

Production Editor:
Gareth Funning
Handling Editor:
Hongyu Sun
Copy & Layout Editor:
Abhineet Gupta

Received:
September 30, 2024
Accepted:
July 25, 2025
Published:
August 19, 2025

1 Introduction

Full-Waveform Inversion (FWI) of seismic data is a technique that allows for the estimation of high-resolution subsurface models. It is formulated in its classic form as the minimization of a misfit function defined as the fit to the data through the l-2 norm of the residuals and tends to be a highly nonlinear inverse problem (Virieux and Operto, 2009; Bunks et al., 1995). Typically, such minimization is achieved using local optimization techniques, as global optimization methods are hindered by the large number of model parameters involved. The technology was initially pioneered by Lailly and Bednar (1983) and Tarantola (1984), and later reintroduced by Pratt (1999). Since its reintroduction in the late nineties, significant research efforts have been directed towards addressing its recognized limitations, particularly focusing on mitigating the cycle skipping phenomenon (Li and Demanet, 2016; Hu et al., 2018). These efforts, combined with the remarkable growth of computational power, have enabled the technique to advance to the point where its application in real data has become both technically and commercially feasible (Michell et al., 2017; Shen et al., 2017; Wang et al., 2019; Li et al., 2023). Despite this achievement, ongoing research continually seeks further enhancements and refinements, leveraging a variety of software sources including in-house proprietary, commercial solutions, and open-source initiatives. In this latter realm, Python prominently stands out as the primary choice for software development. It emerges as a formidable ally to geoscience professionals and students (undergraduate or graduate) due to its versatility, ease of use, and robust library support. Its open-source nature also fosters collaboration and knowledge-sharing within the scientific community, making it an ideal platform for advancing research in different fields of geosciences. With an extensive collection of scientific computing libraries, including several geoscience-related ones (see https://github.com/softwareunderground/awesome-open-geoscience), Python provides essential tools for developing FWI implementations. Simply put, achieving a functional implementation of a traditional FWI scheme, requires the integration of two main building blocks: a wave propagator tool for both forward and adjoint wave equations, alongside an optimization framework that drives the iterative estimation process. As an outstanding example in the first block we can cite the Devito package (Louboutin et al., 2019), a domain-specific language for implementing high-performance finite-difference partial differential equation solvers. On the other hand, the second block presents multiple alternatives, which offer abundant possibilities. Moreover, there exist frameworks designed to seamlessly integrate other tools from these two categories, streamlining FWI workflows (Farris et al., 2023; Thrastarson et al., 2022), as well as Python-based FWI packages (Mardan et al., 2023; Modrak et al., 2018; Hewett and Demanet, 2017).

This article reviews some of the options available for the second block, with a particular focus on those

*Corresponding author: oscar.ladino@fieb.org.br that employ local gradient optimization methods. Despite the variety of frameworks, there remains a lack of comprehensive comparative studies. Such analyses are crucial for helping researchers and practitioners select the most suitable tool for their specific needs, while also identifying gaps where current solutions may fall short. We examine `scipy.optimize` (Virtanen et al., 2020), sotb-wrapper (Mojica, 2022), NLopt (Johnson, 2007), and PyROL by implementing a 2D time-domain FWI. We choose a 2D time-domain FWI experiment because it is simple enough to illustrate the applicability of the freely available optimization software to address the inversion. While this choice highlights simplicity, it's worth noting that the most of the algorithms in the optimization frameworks are implemented in efficient, low-level languages like C, C++ and Fortran, making them equally applicable to more complex 3D scenarios. Although optimization frameworks other than these four exist (note that there are also various deep learning frameworks with gradient optimization methods available), we are restricting ourselves to the four to ensure manageable analysis. These four open-source frameworks offer powerful capabilities for serious multivariate optimization problems. Our aim is to propose a comprehensive comparison between the aforementioned optimization frameworks and provide guidance to practitioners for when to use or not use a particular framework.

2 Optimization frameworks

2.1 scipy.optimize

`scipy.optimize` is a sub-module within the SciPy library, which is a fundamental package for scientific computing in Python. `scipy.optimize` provides a collection of functions for minimizing (or maximizing) objective functions, possibly subject to constraints. It also includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programming, constrained and nonlinear least-squares, root finding, and curve fitting [1].

Key features: The module offers local search via the `minimize()` function, which takes as input the objective function that is being minimized and the initial guess from which to start the search and returns an `OptimizeResult` object that summarizes the success or failure of the search and the details of the solution if found. Optional arguments can be provided, such as the bounds on the input variables, functions for computing the first and second derivatives of the function, and any constraints on the inputs. The "method" parameter of the `minimize()` function enables the selection of a particular optimization approach for the local search. A variety of widely-used local search algorithms are available, including:

- Nelder-Mead Algorithm (`method='Nelder-Mead'`).
- Newton's Method (`method='Newton-CG'`).
- Powell's Method (`method='Powell'`).
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) Algorithm (`method='BFGS'`).
- l-BFGS-B Algorithm (`method='L-BFGS-B'`), which is a limited memory version of BFGS that allows the incorporation of "box" constraints.

The Nelder-Mead, Powell, l-BFGS-B, and Truncated Newton (TNC) methods are viable options for optimization problems with simple bound constraints, with Nelder-Mead and Powell specifically not relying on gradient information. Note that the Conjugate Gradient (CG) method, widely used for FWI, is also available in SciPy; however, the version provided is limited to unconstrained minimization. A comprehensive list of these algorithms and their characteristics can be found in the SciPy documentation at https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize. The line search used in l-BFGS-B is the More-Thuente method (Moré and Thuente, 1994), which seeks to satisfy the Wolfe conditions through a series of polynomial interpolation steps. The line search in TNC uses a safeguarded cubic polynomial interpolation (Gill and Murray, 1979; Nash, 1985).

Documentation: The module provides comprehensive documentation with detailed explanations of each optimization algorithm, including usage examples and guidelines for choosing appropriate methods for different types of problems.

2.2 sotb-wrapper

Sotb-wrapper (Mojica, 2022) is a Python binding implementation of the Seiscope optimization toolbox library (Métivier and Brossier, 2016), that provides a complete coverage of Seiscope toolbox Fortran library API in Python. Python bindings to the Seiscope toolbox were created using ctypes[2]. The ctypes bindings provide direct access from Python to the gradient-based subroutines in the Seiscope optimization toolbox.

Key features: The sotb-wrapper module contains a main `sotb_wrapper` class that includes functions incorporating both first-order methods (steepest-descent and nonlinear conjugate gradient) and second-order methods (limited memory BFGS or l-BFGS and Truncated Newton) for solving large-scale nonlinear optimization problems. These implementations are bolstered by an efficient line-search strategy to ensure robustness. Sotb-wrapper has been designed to mirror the functionality of Seiscope optimization toolbox in Python, facilitating seamless integration with any computational code requiring the resolution of such large-scale minimization problems. Examples in geophysics include traveltime tomography, least-squares migration and FWI. Sotb-wrapper ensures the same benefits as the original toolbox. Firstly, it maintains its key feature: the principle of the reverse communication protocol, which separates routines related to the problem's

[1] https://docs.scipy.org/doc/scipy/reference/optimize.html

[2] https://docs.python.org/3/library/ctypes.html

physics from those concerned with minimization. Following this principle, the optimization routine communicates with the user-defined function, which manages the minimization process through a *do while* loop. At each iteration, the routine requests quantities such as the objective function, gradient, or Hessian-vector product, based on an input/output variable. The values of this variable determine the specific actions to be taken, which vary depending on the choice of optimization routine. This separation provides greater flexibility in code development and maintenance, as changes can be made to either the physics or optimization components independently. Moreover, the framework retains the ability to easily switch between different optimization algorithms, reducing the complexity of implementing second-order methods and ultimately improving computational efficiency, just as in the original toolbox. An important feature of the Seiscope toolbox, as well as the sotb-wrapper, is the inclusion of preconditioned versions of the optimization algorithms. The sotb-wrapper includes four distinct optimization methods, with separate implementations for preconditioned and non-preconditioned versions in certain cases:

- For Steepest Descent and Nonlinear Conjugate Gradient, single functions support both preconditioned and non-preconditioned variants.
- For l-BFGS and Truncated Newton methods, preconditioned and non-preconditioned versions are implemented as separate functions. This results in a total of six functions available.

Because the FWI problem is poorly conditioned, gradient-based methods suffer from a slow convergence rate unless preconditioning techniques are employed. Preconditioning is typically achieved by creating an easily invertible approximation of the Hessian matrix. One of the simplest forms of this approximation involves using only the diagonal elements of the Hessian. A fairly common choice to approximate the diagonal elements of the Hessian is by means of the pseudo-Hessian approach proposed by Shin et al. (2001).

Documentation: Sotb-wrapper has basic tutorials on its GitHub repository that are well-suited for novice users. Its API comes with comprehensive documentation provided through docstrings. These can be accessed conventionally using the `help()` command at the interactive Python prompt upon instantiating a `sotb_wrapper` object. For instance, you can type "help(sotb.PSTD)" to access documentation specific to the preconditioned steepest-descent method.

2.3 PyROL

PyROL, available at https://github.com/trilinos/Trilinos/tree/master/packages/rol/pyrol, serves as an interface between Python and the Sandia National Laboratory's Rapid Optimization Library (ROL). ROL (Kouri et al., 2022) stands as a robust C++ library designed for numerical optimization tasks. Offering a wide array of cutting-edge optimization algorithms, ROL seamlessly integrates into diverse applications. Its versatile programming interface accommodates various computational hardware, including heterogeneous many-core systems featuring both digital and analog accelerators. ROL has been used with success across a spectrum of domains, such as optimal control, optimal design, inverse problems, image processing, and mesh optimization. It's worth noting that there is a separate project available at https://github.com/angus-g/pyrol, which also aims to facilitate the use of the Trilinos ROL library from Python.

Key features: ROL's core design revolves around its abstract linear algebra interface, primarily implemented through the `Vector` class. This feature enables the utilization of any ROL algorithm with various data types, such as C++ `std::vector`, MPI-parallel data structures (e.g., Epetra, Tpetra, PETSc, HYPRE vectors), and GPU data structures (e.g., Kokkos, ArrayFire), among others. Notably, all ROL algorithms operate in a matrix-free manner, relying on user-defined applications of linear and nonlinear operators to vectors, as well as their inverses, through ROL's functional interface. Additionally, ROL offers a specialized middleware called SimOpt for simulation-constrained optimization. In practice, users initially define vectors and functions, then an optimization problem, and utilize optimization solvers within ROL's algorithmic interface to tackle optimization challenges effectively. ROL also offers an extensive repertoire of established and innovative algorithms for smooth optimization. It categorizes optimization problems into four fundamental types: Type U for unconstrained problems, Type B for problems with bound constraints, Type E for problems with equality constraints, and Type G for problems with general constraints. Each problem type may also incorporate linear constraints.

Documentation: As of now, comprehensive documentation for the Python interface is still in progress. However, users can make use of the C++ examples located in the source directory for practical guidance. These examples, located at Trilinos/packages/rol/tutorial, provide demonstrations of various optimization scenarios, including unconstrained optimization, constrained optimization, and simulation-constrained optimization using ROL's SimOpt interface. While maintainers aimed for a release of a user guide by the end of 2023, it seems it is not yet available.

2.4 NLopt

The NLopt library (Johnson, 2007), short for Non-Linear Optimization, is an open-source library for nonlinear optimization. It provides a wide range of algorithms for solving constrained and unconstrained optimization problems, including (non)gradient-based local and global optimization methods. NLopt is written in C, but it includes interfaces for various programming languages, including Python.

Key features: NLopt provides a versatile interface accessible from a large array of programming languages. With this unified interface, users can seamlessly switch between different algorithms by adjusting a single parameter. It supports large-scale optimization tasks, with certain algorithms capable of handling millions of parameters and thousands of constraints. NLopt offers a wide array of algorithms for both global and local optimization, catering to diverse optimization needs. These algorithms range from derivative-free methods relying solely on function values to those utilizing user-supplied gradients. For instance, MMA (Method of Moving Asymptotes), SLSQP, and Low-storage BFGS are some of the options available for local gradient-based optimization, while DIRECT, CRS (Controlled Random Search), and MLSL (Multi-Level Single-Linkage) are popular choices for global optimization. Additionally, NLopt encompasses algorithms suitable for unconstrained optimization, bound-constrained optimization, as well as general nonlinear inequality and equality constraints. In https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/, a comprehensive list of available algorithms is provided, along with descriptions, links to the original code, and references to the corresponding papers, including details on which algorithms support specific types of constraints. As free/open-source software under the GNU LGPL, NLopt provides accessibility and flexibility for various applications.

Documentation: NLopt offers extensive documentation[3], which includes in-depth explanations of the Python API[4]. Additionally, it offers a tutorial that demonstrates the library's functionality across various programming languages through straightforward examples.

3 Numerical experiment

We ran a low-frequency FWI using the Marmousi2 model (Martin et al., 2006), which was discretized into 88×426 grid points with a 40-m grid interval. We smoothed both the true and the initial models, since it was a low-frequency experiment (Fig. 1). Seismic modeling was performed to produce experimental synthetic data using a Ricker wavelet with a peak frequency of 4 Hz. We used 16 shots and 426 receivers evenly distributed across surface of the model. FWI with box constraints was executed separately for each framework employing the l-BFGS algorithm, which is common across all packages and tailored for the specific constrained optimization problem at hand. The limited memory BFGS (l-BFGS) is one of the popular methods for solving the FWI problem (Luo et al., 2023; Shoja et al., 2018; Fabien-Ouellet et al., 2017). Its implementation is almost identical to that of the standard BFGS method, the only difference is that the inverse Hessian approximation is not formed explicitly, but defined by a small number of BFGS updates. We refer to Nocedal (1980) and Liu and Nocedal (1989b) for a detailed review of the l-BFGS method, along with the more comprehensive discussion in Nocedal and Wright (1999).

In the subsequent subsections, we provide a general overview of the steps involved in setting up and solving FWI using each framework, along with abbreviated code snippets illustrating these steps. In the code snippets, ellipses (. . .) have been strategically in-

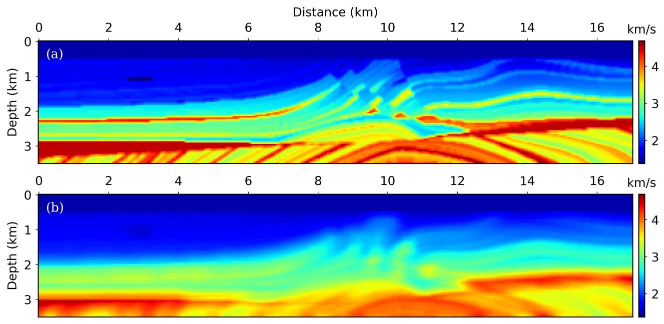

Figure 1 (a) Marmousi2 model. (b) Initial model.

cluded to omit certain instructions due to space constraints. However, these omissions do not compromise the understanding of the presented code syntax. The complete codes are available at https://github.com/ofmla/optim_python_fwi. Reader is invited to refer to the repository to see these code snippets in the context of the full FWI implementation, as well as to consult the documentation of each framework to explore additional customization options and advanced functionalities.

3.1 scipy.optimize

According to the documentation of `scipy.optimize.minimize`, various options exist for performing bounds-constrained minimization. By selecting the l-BFGS-B method (Byrd et al., 1995), FWI can be efficiently addressed by simply invoking the `minimize()` function, using the `bounds` parameter. Initially, we define a function that returns both the objective function and its gradient, computed through the adjoint-state method via Devito. In this specific function design, the `jac` parameter, which is an optional argument to `minimize()`, must be a Boolean, configured to "True". It's important to note that this function, along with the initial guess, are mandatory arguments for `minimize()`. Additionally, l-BFGS-B accepts a callback function, enabling operations on the solution after every iteration. This feature can be utilized, for instance, to monitor the true relative solution error in experiments involving synthetic models. Another important aspect of l-BFGS-B in SciPy is that it expects the gradient to be provided as a flat array composed of 64-bit floats. Therefore, even though the forward and adjoint computations in the function are done in float32, casting the gradient to float64 is required. Code for solving FWI by using `scipy.optimize.minimize` is shown in Listing 1. Note the `options` parameter, a dictionary of solver options where we set `maxiter`, an integer denoting the maximum number of iterations to execute and `disp`, a boolean flag that enables to receive convergence notifications. The outcome of the optimization is encapsulated within `result`, an `OptimizeResult` object. The attribute `x` attached to it is the array representing the solution. One of the l-BFGS-specific options is the parameter m that determines the number of BFGS corrections saved. In l-BFGS-B, this parameter corresponds to the `maxcor` variable, listed within the `options` dictionary. Users can view detailed

[3]https://nlopt.readthedocs.io/en/latest/
[4]https://nlopt.readthedocs.io/en/latest/NLopt_Python_Reference/

information about the key-value pairs in the options dictionary for a specific optimization method using the show_options function. For example, calling scipy.optimize.show_options(solver='minimize', method='L-BFGS-B') displays additional options available for the l-BFGS-B method. Note that, besides specifying maxiter, other stopping criteria can also be configured in the options dictionary.

Listing 1 Syntax for solving FWI problem with scipy.optimize. As of the submission of this manuscript, the SciPy version is 1.14.1. In later versions of SciPy, the disp keyword has been deprecated.

```
from scipy.optimize import minimize
import numpy as np

# Initial guess
v0= get_vp(...) # Get start vp - hdn. impl.
x=1.0/(v0.reshape(-1).astype(np.float32))**2

# Define bbox constrs. on the solution.
vmin=1.4
vmax=4.0
bounds=[(1.0/vmax**2, 1.0/vmin**2) for _ in
        range(x.size)]

out = minimize(loss, x, jac=True,
               method='L-BFGS-B',
               bounds=bounds,
               options={'disp': True,
                        'maxiter': 20,
                        'maxcor':10})
```

3.2 sotb-wrapper

All optimization algorithms within sotb.wrapper allow you to specify bounds for the variables. Each function comes with parameters called lb and ub, which let you define lower and upper bounds for the solution values. By default, if users do not specify any bounds, the None value is used. Unlike scipy.optimize, where solving FWI is as simple as making a function call, here users have a bit more responsibility in designing the optimization process. To begin, users create an instance of a sotb_wrapper class. Then, they compute the cost and gradient associated with their initial guess. The initial cost along with the maximum iteration number parameters are mandatory in the set_inputs function, which sets up the optimization parameters. Once the parameters are configured, users need to set up an exit-controlled optimization loop. This loop continuously calls an optimization solver as long as a flag variable remains different from 2 and 4. A value of 2 indicates successful convergence of the minimization process, while a value of 4 indicates a failure in the line-search process. This flag variable is part of the reverse communication protocol, dictating what actions the minimization routine should take. It must be initialized to 0 before usage in the algorithm routine within the optimization loop. A flag value of 1 indicates that the line-search process is ongoing and necessitates a fresh evaluation of the objective function along with its gradient to proceed. The implementation of FWI using sotb-wrapper is demonstrated in Listing 2. Within the l-BFGS method in sotb-wrapper, if the l parameter is not specified in the set_inputs function, the default number of factors (most recent vector pairs) utilized to implicitly represent the inverse Hessian is set to 10.

Listing 2 Syntax for solving FWI problem with sotb-wrapper.

```
import numpy as np
from sotb_wrapper import interface

# Initial guess and bbox constraints
v0= get_vp(...) # Get start vp - hdn. impl.
x=1.0/(v0.reshape(-1).astype(np.float32))**2
# Dimension and first flag initialization
n,flag = x.size,0

vmin,vmax = 1.4,4.0  # Min and max vp values
# Lower and upper bound arrays
lb=np.full(n, 1.0/vmax**2, dtype=np.float32)
ub=np.full(n, 1.0/vmin**2, dtype=np.float32)

# Create an instance of the SEISCOPE
    optimization toolbox (sotb) Class.
sotb = interface.sotb_wrapper()

# Define fields of the UserDefined derived
    type in Fortran (ctype structure).
print_flag, debug = 1, False
niter_max, l = 20, 10

# Computation of the cost and gradient
    associated with the initial guess
fcost, grad = loss(x, ...) # hdn. impl.

# parameter initialization
sotb.set_inputs(
    fcost, niter_max,
    print_flag=print_flag,
    l=l, debug=debug
)

while flag != 2 and flag != 4:
    flag = sotb.LBFGS(n, x, fcost, grad,
                      flag, lb, ub)
    if flag == 1:
    # compute cost and gradient at point x
        fcost, grad = loss(x, ...)
```

3.3 PyROL

PyROL allows for solving bound-constrained problems (Type B) using various methods, including projected gradient and projected Newton methods, as well as primal-dual active set methods. Implementing FWI with PyROL is more complex and requires several steps, but ultimately FWI is solved by calling the solve() method from the Solver class. Users must define the initial guess x (as a NumpyVector object) and create a subclass of the Objective class. This subclass includes its own versions of the value and gradient methods. The value method computes the cost associated with x throughout the optimization process, while the gradient method calculates the corresponding gradient. Following these definitions, the user must create instances of the subclass of FWIObjective and Bounds class. The Bounds class is instantiated by providing two arguments: the `lower` and the `upper` bounds on `x`, i.e., `bnd = Bounds(lower, upper)`, where `lower` and `upper` are both `NumpyVector` objects. The final step involves creating an optimization problem via the `Problem` class and then add the bounds with its `addBoundConstraint` method. This problem can then be resolved using the `solve` method of an instance of the `Solver` class. The code block in Listing 3 illustrates the implementation of FWI using PyROL.

Listing 3 Syntax for solving FWI problem with PyROL.

```
import numpy as np
from pyrol import (getCout, Objective,
    Problem, Solver, Bounds)
from pyrol.pyrol.Teuchos import ParameterList
from pyrol.vectors import NumPyVector

class FWIObjective(Objective):
    def __init__(self, ...):
        ... # hdn. impl.
        super().__init__()

    def value(self, x, tol):
        f = ... # hdn. impl.
        return f

    def gradient(self, g, x, tol):
        g[:] = ... # hdn. impl.

# Initial guess and box constraints
v0 = get_vp(...) # Get start vp - hdn. impl.
x = NumPyVector(np.array(1.0/(v0.reshape(-1).
    astype(np.float32))**2))

lower = NumPyVector(np.full(n, 1./vmax**2,
    dtype=np.float32))
upper = NumPyVector(np.full(n, 1./vmin**2,
    dtype=np.float32))

# Configure parameter list
params = build_parameter_list()

# Set the output stream.
stream = getCout()

# Set up the FWI problem
objective = FWIObjective(...)
bnd = Bounds(lower, upper)
problem = Problem(objective, x)
problem.addBoundConstraint(bnd)

# Solve
solver = Solver(problem, parameters)
solver.solve(stream)
```

3.4 NLopt

NLopt comprises various gradient-based optimization algorithms, categorized by the named constants L (for local) and D (for gradient-based). In this case, we opted for l-BFGS algorithm (Liu and Nocedal, 1989a), denoted in NLopt's terminology as `NLOPT_LD_BFGS`. The core of the NLopt API centers on an instance of the `nlopt.opt` class. Through the methods provided by this object, users define all the optimization parameters, including dimensions, algorithm selection, stopping criteria, constraints, objective function, and more. The user is required to create an instance variable `opt` of the `nlopt.opt` class with algorithm and dimensionality of the problem passed as arguments. Then define a function `f`, which takes two arguments, `x` and `grad` (the gradient of the cost function with respect to the optimization parameters at `x`). It modifies `grad` in-place and also provides the value of the function at the point `x`. To minimize the objective function, one should specify the objective function by calling `opt.set_min_objective(f)`. Additionally, set the lower bounds with `opt.set_lower_bounds(lb)` and the upper bounds with `opt.set_upper_bounds(ub)`, where "lb" and "ub" are arrays of length "n", matching the dimension passed to the `nlopt.opt` constructor. Listing 4 displays the code demonstrating FWI resolution with the use of NLopt. Different stopping criteria can be utilized. In the code block in Listing 4, the maximum number of function evaluations is set via the `set_maxeval` method. The number `l` of stored vectors utilized to implicitly represent the inverse Hessian is set with `opt.set_vector_storage(l)`. Finally, the optimization process is initiated by invoking the `opt.optimize` method.

Listing 4 Syntax for solving FWI problem with NLopt.

```
import numpy as np
import nlopt

def myfunc(x, grad):
    if grad.size > 0:
        fcost, grad[:] = loss(x) # hdn. impl.
    return np.float64(fcost)

# Initial guess and bbox constraints
v0 = get_vp(...) # Get start vp - hdn. impl.
x = 1.0/(v0.reshape(-1).astype(np.float32))
    **2
grad = np.zeros_like(x, dtype=np.float32)

vmin,vmax = 1.4,4.0  # Min and max vp values
# Lower and upper bound arrays
lb = np.full(n, 1.0/vmax**2, dtype=np.float32
    )
ub = np.full(n, 1.0/vmin**2, dtype=np.float32
    )

opt = nlopt.opt(nlopt.LD_LBFGS, int(x.size))
opt.set_min_objective(myfunc)
opt.set_lower_bounds(lb)
opt.set_upper_bounds(ub)
opt.set_maxeval(35)
opt.set_vector_storage(10)

# Optimization
minx = opt.optimize(x)
```

4 On the FWI results

The reconstructed FWI models obtained by using the l-BFGS algorithm available in the different optimization frameworks are shown in Fig. 2. The prevalent termination criterion among all frameworks, with the exception of NLopt, is the maximum number of iterations, which was set to 20 iterations. The number m of correction pairs stored was set to 10 in all cases. Additionally, parameters that were not available in all frameworks or that users could not explicitly set were left at their default values. To ensure a fair comparison with NLopt, we set its termination criterion based on the highest number of function evaluations observed among the other frameworks. Specifically, we considered the maximum number of function evaluations performed by the sotb-wrapper l-BFGS, which also imposes weak Wolfe conditions in its line search strategy. As a result, the NLopt optimization process was set to stop after 34 function evaluations. SciPy's l-BFGS-B uses a line search method based on cubic interpolation, which determines a step length that satisfies the strong Wolfe conditions—ensuring that the gradient magnitude decreases sufficiently. In contrast, the l-BFGS implementation in the NLopt library attempts to find a point that satisfies the weak Wolfe conditions by using cubic interpolation (Luksan et al., 2007). The l-BFGS implementation in PyROL employs a simple backtracking line search that only enforces the sufficient decrease condition. Meanwhile, the sotb-wrapper l-BFGS computes step lengths that satisfy the weak Wolfe conditions using a bracketing strategy proposed in Bonnans et al. (2003). Fig. 3 illustrates the convergence history, displaying the normalized objective function's value against the count of function evaluations. It is noteworthy that, excluding the NLopt library, the data presented in the figure correspond to the cumulative number of function evaluations per iteration for the other libraries. In these cases, each iteration consistently produces a lower function value than its predecessor, indicative of a descending trend. However, fluctuations in function values may occur within each iteration, primarily attributable to line search techniques. This variability is particularly evident in the NLopt case, where individual function evaluations exhibit fluctuations, especially during the initial stages of the inversion process. In terms of computational cost, the l-BFGS implementation in PyROL is more expensive than initially expected, despite using only the sufficient decrease condition. This is because the computation of the objective function is decoupled from the gradient calculation, meaning the forward pass used to compute the objective function is not reused in the gradient computation. As a result, PyROL requires 21 gradient evaluations (forward and adjoint) plus an additional 28 forward computations: 8 in the first iteration and at least 1 in subsequent iterations. In contrast, the l-BFGS implementations in the sotb-wrapper, NLopt, and SciPy frameworks are more efficient, as they take advantage of a function that simultaneously computes both the objective function and its gradient, eliminating the need for additional forward passes. The sotb-wrapper and NLopt implementations required 34 gradient evaluations (with the stopping criterion for the NLopt l-BFGS set to match the number of function evaluations performed by the sotb-wrapper l-BFGS). Meanwhile, SciPy's l-BFGS-B implementation computed 26 gradient evaluations, making it the fastest in terms of gradient evaluations among the compared methods. Although the line-search algorithms in PyROL, SciPy, and sotb-wrapper required multiple internal iterations in the first minimization step to adjust the step length, SciPy's l-BFGS-B still outperformed the other methods in terms of gradient evaluations, making it the most computationally efficient among the frameworks tested. The inversions took an average of 04:01, 05:20, 05:52, and 06:20 minutes when using the `scipy.optimize`, sotb-wrapper, NLopt, and PyROL frameworks, respectively. All experiments were conducted on a Dell workstation equipped with an Intel(R) Xeon(R) E5-1607 processor running at 3.00 GHz with 16 GB of RAM. The processor has four cores, and the operating system was Ubuntu 22.04. Shots were distributed across the four cores during the inversions. While our comparison primarily examines specific characteristics of optimization packages rather than the methods themselves, it's notable that all tested l-BFGS implementations successfully reconstructed acceptable models. However, in this particular case, it's worth emphasizing that SciPy's l-BFGS-B method produced the most accurate final result.

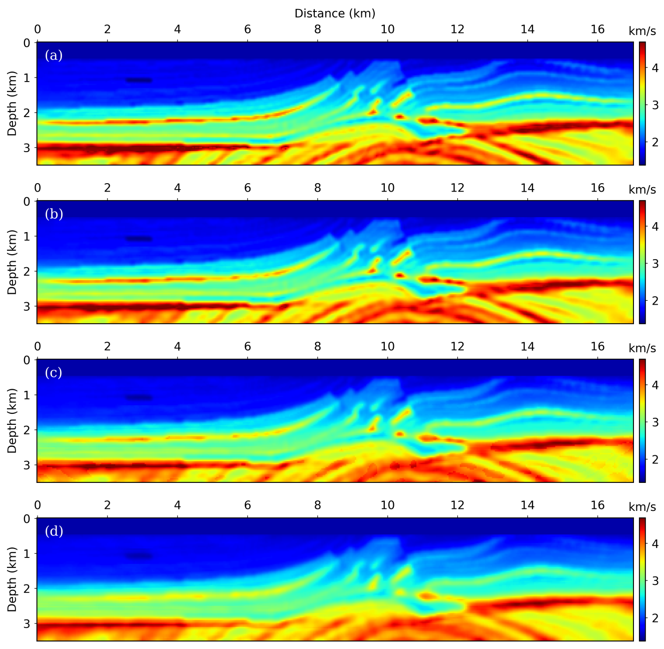

Figure 2 Estimated velocity model using different optimization frameworks. (a) `scipy.optimize`, (b) sotb-wrapper, (c) PyROL, and (d) NLopt.

5 Discussion and conclusions

We examine and discuss various open-source optimization software tools tailored for FWI (although they can also be used for the inversion of various geophysical data), all of which are built with Python interfaces. These include `scipy.optimize`, sotb-wrapper, PyROL, and NLopt. Additionally, we assessed the inversion capabilities of each package through a simple numerical experiment using the Marmousi2 model and employing identical input data and algorithms.

PyROL distinguishes itself as the most comprehensive framework due to its extensive collection of cutting-edge optimization algorithms. Its versatility is evident

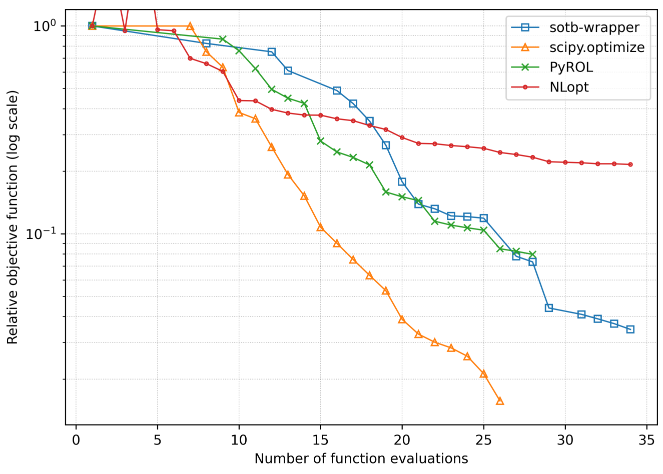

Figure 3 Evaluation of the objective function for four different FWI tests shown in Fig. 2

in its successful application across a wide range of fields, including geophysics. `scipy.optimize` is also a valuable resource for anyone working on optimization problems in Python, offering a wide range of methods to suit different scenarios, such as constrained optimization. NLopt provides a unified interface for various optimization routines available online, alongside original implementations of different algorithms. It supports Python and also offers a wide range of optimization algorithms for local optimization tasks. Sotb-wrapper provides Python bindings for Seiscope Optimization toolbox, featuring gradient-based algorithms tailored for large-scale geophysical problems.

In terms of documentation `scipy.optimize` and NLopt stand out. `scipy.optimize` has extensive and detailed resources with simple examples. In contrast, while the NLopt framework boasts comprehensive documentation, it only offers a basic Python example. Sotb-wrapper has basic tutorials on its GitHub repository that are well-suited for novice users. PyROL's documentation website is under construction and not yet available, but basic examples on its repository can assist users in gaining a better understanding of the package's features.

When it comes to the learning process, `scipy.optimize`'s documentation proves to be a valuable resource, enabling a swift and efficient start. Sotb-wrapper's examples comprehensively encompass the necessary elements for solving optimization problems with the package, making it remarkably user-friendly. On the other hand, while PyROL offers an impressive array of features, its lack of documentation can prove to be a hurdle for beginners, demanding extra perseverance to fully comprehend and harness its capabilities. Regarding NLopt, although there are not many examples, given its simplicity, understanding how it works and maximizing its potential is not very difficult.

Our numerical experiment demonstrated accurate model resolution across all tested optimization packages (refer to Fig. 2). However, variations arose due to the distinct characteristics of each package. This, along with the fact that algorithms, even if they were the same, had been implemented differently, led to the aforementioned differences.

An important aspect not explicitly covered in our comparison is the evolution and maintenance of the optimization frameworks. Among the four Python-based gradient-based optimization libraries analyzed—`scipy.optimize`, NLopt, PyROL, and sotb-wrapper—three are actively maintained and continue to evolve, while one has remained largely static. `scipy.optimize` benefits from being part of the broader SciPy ecosystem, which is a fundamental library in scientific computing with continuous contributions from a large developer community. NLopt, despite being an independent project, has gained substantial popularity across various scientific and engineering fields, leading to ongoing updates and improvements. PyROL, as a Python wrapper for ROL, has a clear path for continued development and enhancement. While its own updates may not be as frequent, PyROL benefits from ROL's ongoing maintenance and improvements, given that ROL is a well-established optimization package actively developed by a U.S. national laboratory. These three frameworks, being part of larger, well-recognized projects, are expected to receive continued support and refinements over time. In contrast, sotb-wrapper has seen limited recent development, as its functionalities are well-established and were specifically designed with large-scale geophysical optimization problems in mind. Unlike the other three frameworks, which are widely used across multiple disciplines, sotb-wrapper remains known within the geophysics community. Nonetheless, while it may not exhibit the same level of ongoing evolution, its existing capabilities remain relevant for the problems it was designed to address. Moreover, as open-source libraries, they are inherently designed for extensibility, allowing users with programming expertise to modify and distribute the code in accordance with their licenses, thereby adding new features, enhancing existing ones, or developing entirely new applications tailored to specific requirements.

Beyond maintainability, another crucial aspect to consider is the scalability of these frameworks for handling large-scale problems efficiently. While our study focused on a simplified 2D example, nothing inherently restricts these optimization frameworks from being applied to larger 3D experiments. The methodologies and implementations presented can be directly adapted to higher-dimensional problems. In our experiments, we utilized Dask (Rocklin, 2015) and the Distributed Dask library to efficiently distribute computations across cores in a workstation, demonstrating that these frameworks can already leverage parallel execution. For 3D inversions, one could integrate parallel computing frameworks such as Dask-Jobqueue (https://github.com/dask/dask-jobqueue) to scale computations across multiple nodes in an HPC environment.

Although comparative studies like ours are important, they are rare. FWI practitioners who wish to utilize Python optimization packages for their research or applications are confronted with a challenging decision due to the various available options and the limited scientific evaluation conducted by the research community to compare them. Our study aimed to simplify this decision-making process and provide guidance to practitioners in selecting the package that best suits their needs.

Acknowledgements

The author would like to thank CENPES/Petrobras and ANP for their support of this study under the LDE project at Senai Cimatec. This work is also partially supported by the National Institute of Science and Technology of Petroleum Geophysics (INCT-GP). The author gratefully acknowledges the contribution of OpenAI's ChatGPT for refining the sentence structure and enhancing the overall readability of this manuscript.

Data and code availability

The data used in this study is licensed under the Creative Commons Attribution 4.0 International License. It is part of the SEG Open Data collection and can be downloaded from https://s3.amazonaws.com/open.source.geoscience/open_data/elastic-marmousi/elastic-marmousi-model.tar.gz. The code is fully open-source and available at https://github.com/ofmla/optim_python_fwi.

Competing interests

The author declares that he has no competing interests.

References

Bonnans, J. F., Gilbert, J. C., Lemaréchal, C., and Sagastizábal, C. A. *Numerical Optimization: Theoretical and Practical Aspects*. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-662-05078-1.

Bunks, C., Saleck, F. M., Zaleski, S., and Chavent, G. Multiscale seismic waveform inversion. *GEOPHYSICS*, 60(5):1457–1473, Sept. 1995. doi: 10.1190/1.1443880.

Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, Sept. 1995. doi: 10.1137/0916069.

Fabien-Ouellet, G., Gloaguen, E., and Giroux, B. A stochastic L-BFGS approach for full-waveform inversion. In *SEG Technical Program Expanded Abstracts 2017*, page 1622–1627. Society of Exploration Geophysicists, Aug. 2017. doi: 10.1190/segam2017-17783222.1.

Farris, S., Barnier, G., Biondi, E., and Clapp, R. Pyseis: A high-performance, user-friendly Python package for GPU-accelerated seismic modeling and subsurface imaging, Dec. 2023. doi: 10.1190/image2023-3916155.1.

Gill, P. E. and Murray, W. *Conjugate-Gradient Methods for Large-Scale Nonlinear Optimization.* Defense Technical Information Center, Oct. 1979. doi: 10.21236/ada078713.

Hewett, R. J. and Demanet, L. PySIT: Seismic imaging toolbox for Python. *Mass. Inst. Technol., Cambridge, MA, USA, Tech. Rep*, 2017. http://pysit.org.

Hu, W., Chen, J., Liu, J., and Abubakar, A. Retrieving Low Wavenumber Information in FWI: An Overview of the Cycle-Skipping Phenomenon and Solutions. *IEEE Signal Processing Magazine*, 35(2):132–141, Mar. 2018. doi: 10.1109/msp.2017.2779165.

Johnson, S. G. The NLopt nonlinear-optimization package. https://github.com/stevengj/nlopt, 2007.

Kouri, D., Ridzal, D., von Winckel, G., and Javeed, A. Get ROL-ing: An Introduction to Sandia's Rapid Optimization Library. In *Proposed for presentation at the International Conference on Continuous Optimization held July 24-28, 2022 in Bethlehem, PA United States of America*. US DOE, July 2022. doi: 10.2172/2004204.

Lailly, P. and Bednar, J. The seismic inverse problem as a sequence of before stack migrations. In *Conference on inverse scattering: theory and application*, pages 206–220. Philadelphia, Pa, 1983.

Li, J., Rusmanugroho, H., Kalita, M., Xin, K., and Dzulkefli, F. S. 3D anisotropic full-waveform inversion for complex salt provinces. *Frontiers in Earth Science*, 11, Apr. 2023. doi: 10.3389/feart.2023.1164975.

Li, Y. E. and Demanet, L. Full Waveform Inversion With Extrapolated Low Frequency Data. In *Offshore Technology Conference Asia*, 16OTCA. OTC, Mar. 2016. doi: 10.4043/26626-ms.

Liu, D. C. and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1–3): 503–528, Aug. 1989a. doi: 10.1007/bf01589116.

Liu, D. C. and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1–3): 503–528, Aug. 1989b. doi: 10.1007/bf01589116.

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., and Gorman, G. J. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12 (3):1165–1187, Mar. 2019. doi: 10.5194/gmd-12-1165-2019.

Luksan, L., Matonoha, C., and Vlček, J. New subroutines for large-scale optimization. Technical Report Tech. Rep. V-999, ICS AS CR, Prague, Czech Republic, June 2007. https://asep.lib.cas.cz/arl-cav/en/detail/?&idx=cav_un_epca*0085718.

Luo, J., Zhou, H., Wu, R.-S., and Huang, X. Salt and subsalt structure recovery–An envelope-based waveform inversion with local angle domain illumination compensation and L-BFGS. *GEOPHYSICS*, 88(4):R453–R467, July 2023. doi: 10.1190/geo2022-0550.1.

Mardan, A., Giroux, B., and Fabien-Ouellet, G. Pyfwi: A Python Package for Full-Waveform Inversion and Reservoir Monitoring, 2023. doi: 10.2139/ssrn.4330227.

Martin, G. S., Wiley, R., and Marfurt, K. J. Marmousi2: An elastic upgrade for Marmousi. *The Leading Edge*, 25(2):156–166, Feb. 2006. doi: 10.1190/1.2172306.

Michell, S., Shen, X., Brenders, A., Dellinger, J., Ahmed, I., and Fu, K. Automatic velocity model building with complex salt: Can computers finally do an interpreter's job? In *SEG Technical Program Expanded Abstracts 2017*, page 5250–5254. Society of Exploration Geophysicists, Aug. 2017. doi: 10.1190/segam2017-17778443.1.

Modrak, R. T., Borisov, D., Lefebvre, M., and Tromp, J. SeisFlows–Flexible waveform inversion software. *Computers & Geosciences*, 115:88–95, June 2018. doi: 10.1016/j.cageo.2018.02.004.

Mojica, O. F. sotb-wrapper, 2022. doi: 10.5281/ZENODO.7117744.

Moré, J. J. and Thuente, D. J. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software*, 20(3):286–307, Sept. 1994. doi: 10.1145/192115.192132.

Métivier, L. and Brossier, R. The SEISCOPE optimization toolbox: A large-scale nonlinear optimization library based on reverse communication. *GEOPHYSICS*, 81(2):F1–F15, Mar. 2016. doi: 10.1190/geo2015-0031.1.

Nash, S. G. Preconditioning of Truncated-Newton Methods. *SIAM Journal on Scientific and Statistical Computing*, 6(3):599–616, July 1985. doi: 10.1137/0906042.

Nocedal, J. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980. doi: 10.1090/s0025-5718-1980-0572855-7.

Nocedal, J. and Wright, S. J. *Numerical optimization*. Springer, 1999. doi: 10.1007/978-0-387-40065-5.

Pratt, R. G. Seismic waveform inversion in the frequency domain, Part 1: Theory and verification in a physical scale model. *GEOPHYSICS*, 64(3):888–901, May 1999. doi: 10.1190/1.1444597.

Rocklin, M. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, SciPy, page 126–132. SciPy, 2015. doi: 10.25080/majora-7b98e3ed-013.

Shen, X., Ahmed, I., Brenders, A., Dellinger, J., Etgen, J., and Michell, S. Salt model building at Atlantis with full-waveform inversion. In *SEG Technical Program Expanded Abstracts 2017*. Society of Exploration Geophysicists, Aug. 2017. doi: 10.1190/segam2017-17738630.1.

Shin, C., Jang, S., and Min, D. Improved amplitude preservation for prestack depth migration by inverse scattering theory. *Geophysical Prospecting*, 49(5):592–606, Sept. 2001. doi: 10.1046/j.1365-2478.2001.00279.x.

Shoja, S., Abolhassani, S., and Amini, N. A Comparison between Time-Domain and Frequency-Domain Full Waveform Inversion. In *80th EAGE Conference and Exhibition 2018*, Copenhagen2018. EAGE Publications BV, June 2018. doi: 10.3997/2214-4609.201801667.

Tarantola, A. Inversion of seismic reflection data in the acoustic approximation. *GEOPHYSICS*, 49(8):1259–1266, Aug. 1984. doi: 10.1190/1.1441754.

Thrastarson, S., van Herwaarden, D.-P., and Fichtner, A. Inversionson: Fully Automated Seismic Waveform Inversions, Mar. 2022. doi: 10.31223/x5f31v.

Virieux, J. and Operto, S. An overview of full-waveform inversion in exploration geophysics. *GEOPHYSICS*, 74(6):WCC1–WCC26, Nov. 2009. doi: 10.1190/1.3238367.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, I., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., Vijaykumar, A., Bardelli, A. P., Rothberg, A., Hilboll, A., Kloeckner, A., Scopatz, A., Lee, A., Rokem, A., Woods, C. N., Fulton, C., Masson, C., Häggström, C., Fitzgerald, C., Nicholson, D. A., Hagen, D. R., Pasechnik, D. V., Olivetti, E., Martin, E., Wieser, E., Silva, F., Lenders, F., Wilhelm, F., Young, G., Price, G. A., Ingold, G.-L., Allen, G. E., Lee, G. R., Audren, H., Probst, I., Dietrich, J. P., Silterra, J., Webber, J. T., Slavič, J., Nothman, J., Buchner, J., Kulick, J., Schönberger, J. L., de Miranda Cardoso, J. V., Reimer, J., Harrington, J., Rodríguez, J. L. C., Nunez-Iglesias, J., Kuczynski, J., Tritz, K., Thoma, M., Newville, M., Kümmerer, M., Bolingbroke, M., Tartre, M., Pak, M., Smith, N. J., Nowaczyk, N., Shebanov, N., Pavlyk, O., Brodtkorb, P. A., Lee, P., McGibbon, R. T., Feldbauer, R., Lewis, S., Tygier, S., Sievert, S., Vigna, S., Peterson, S., More, S., Pudlik, T., Oshima, T., Pingel, T. J., Robitaille, T. P., Spura, T., Jones, T. R., Cera, T., Leslie, T., Zito, T., Krauss, T., Upadhyay, U., Halchenko, Y. O., and Vázquez-Baeza, Y. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, Feb. 2020. doi: 10.1038/s41592-019-0686-2.

Wang, P., Zhang, Z., Mei, J., Lin, F., and Huang, R. Full-waveform inversion for salt: A coming of age. *The Leading Edge*, 38(3):204–213, Mar. 2019. doi: 10.1190/tle38030204.1.