

Title**simwave** - A Finite Difference Simulator for Acoustic Waves Propagation**Paper Authors**

1. Souza, Jaime Freire de;
2. Moreira, João Baptista Dias;
3. Roberts, Keith Jared;
4. Gaiosio, Roussian di Ramos Alves;
5. Gomi, Edson Satoshi;
6. Silva, Emílio Carlos Nelli;
7. Senger, Hermes.

Paper Author Roles and Affiliations

1. Federal University of São Carlos (UFSCar). Roles: Software, Validation, Writing - original draft, Writing - review & editing.
2. University of São Paulo (USP). Roles: Numerical methods, Software, Validation, Writing - original draft, Writing - review & editing.
3. University of São Paulo (USP). Roles: Numerical methods, Software, Validation, Writing - original draft, Writing - review & editing.
4. Federal University of São Carlos (UFSCar). Roles: Software, Writing - review & editing.
5. University of São Paulo (USP). Roles: Supervision, Writing - original draft, Writing - review & editing.
6. University of São Paulo (USP). Roles: Funding acquisition, Project administration, Supervision, Writing review.
7. Federal University of São Carlos (UFSCar). Roles: Funding acquisition, Project administration, Supervision, Writing - original draft, Writing - review & editing.

Abstract

simwave is an open-source Python package to perform wave simulations in 2D or 3D domains. It solves the constant and variable density acoustic wave equation with the finite difference method and has support for domain truncation techniques, several boundary conditions, and the modeling of sources and receivers given a user defined acquisition geometry. The architecture of **simwave** is designed for applications with geophysical exploration in mind. Its Python front-end enables straightforward integration with many existing Python scientific libraries for the composition of more complex workflows and applications (e.g., migration and inversion problems). The back-end is implemented in C enabling performance portability across a range of computing hardware and compilers including both CPUs and GPUs.

Keywords

Acoustic waves simulation, seismology, finite differences, high performance computing, Python.

1 Introduction

Acoustic waves are a means of energy propagation through a medium in space. These waves travel with a characteristic velocity and exhibit phenomena like diffraction, reflection and interference as they interact with the medium. The propagation of acoustic waves can be described by pressure variation, particle velocity, particle displacement, and/or acoustic intensity. The propagation of acoustic waves is often used as a remote sensing tool to probe domains that are otherwise difficult to physically observe. Depending on the properties of the medium and the application, the simulation of acoustic waves may or may not consider variations in material density. For example, the acoustic wave equation with a constant density approximation is frequently used in seismic inversion workflows to estimate the P-wave velocity in the ground, which is later used to help locate raw material deposits such as oil and gas [37, 40]. In medical imaging, similar methods are used that consider variations in material density or elasticity to study and diagnose tumors and other lesions in the human body [15, 24, 44]. Acoustic tomography also plays an important role in understanding and monitoring ocean processes such as the global tides and internal waves [10, 29] and atmospheric turbulence [19]. In structural modeling, the acoustic wave can be used to identify failures in complex structures such as bridges and buildings [20, 38].

Many wave propagators are part of comprehensive propriety codes that are developed by companies for industrial-grade workflows. In this context, usually the software is not available to independent researchers. Often many of these industrial workflows require computationally efficient implementations that can be used at many different computing scales, and this implies that re-implementation at some level is required.

`simwave` is a Python package that enables researchers to model acoustic waves propagation using short Python scripts with implementations that are verified and optimized for high performance. To be useful to a wide range of applications, the package is made to be flexible across hardware and software environments. Users interact with `simwave` with a Python application programming interface (API) by passing user inputs that control the desired accuracy of the simulation. Many components of `simwave` are implemented for applications with geophysical exploration and the simulation of waves can occur with either the assumption of constant or a variable density medium.

1.1 Applications

The acoustic wave is often used to solve inversion problems to estimate material properties such as in full waveform inversion (FWI) [12]. These inverse problems are particularly computationally demanding as they require many wave propagation simulations in order to produce meaningful solutions to the inverse problem. As a result, the primary computational cost of the inversion process is proportional to the speed at which one can simulate the propagation of a wave.

An overview of a typical inversion setup is shown in Figure 1: As an example, in a typical FWI setup in a marine environment, a ship tows a cable with hundreds of recording devices termed *receivers* potentially several kilometers long. On the ship,

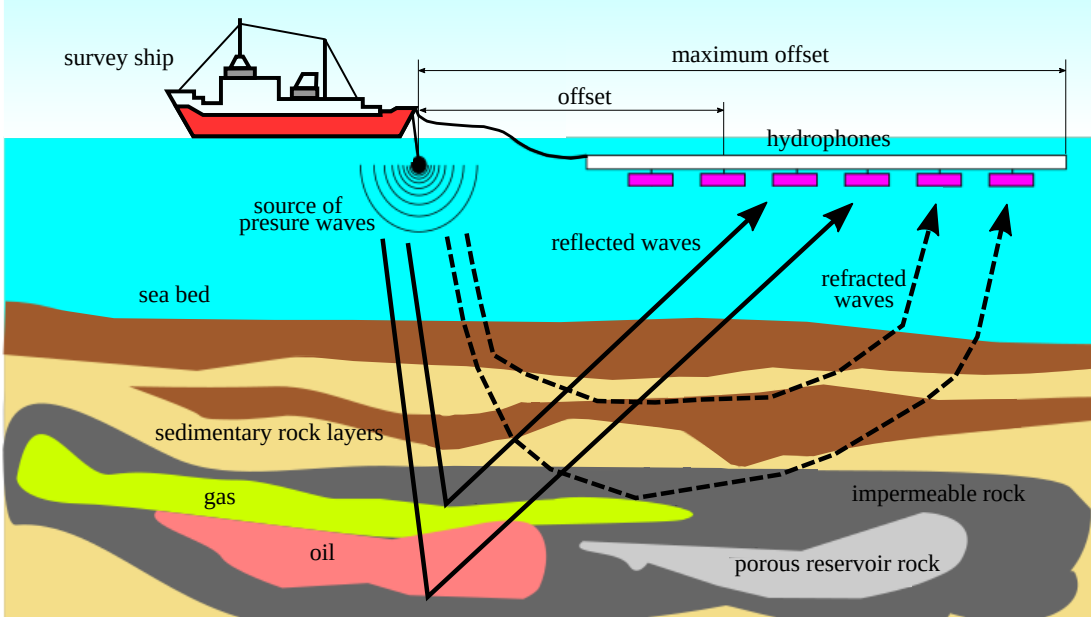


Figure 1: An application of acoustic wave propagation as a remote sensing tool.

small controlled explosions known as a *shots* or *sources* are periodically fired. These sources propagate acoustic waves that interact with the subsurface medium and produce signals recorded by the receivers. The collection of seismic signals for a particular source explosion event is referred to as a *shot record* and the quantity and the location of the sources with respect to the location of the receivers is referred to as an *acquisition geometry*. A similar technique is applied to model how ultrasound energy is transmitted through the skull to generate accurate three-dimensional images of the human brain with sub-millimeter resolution [14].

2 Governing equations

The propagation of mechanical waves can be modeled with the elastic wave equation [12]:

$$\rho(\mathbf{x}) \frac{\partial^2 \mathbf{u}}{\partial t^2}(\mathbf{x}, t) = \nabla \cdot \boldsymbol{\sigma}(\mathbf{x}, t) + \rho(\mathbf{x}) \mathbf{b}(\mathbf{x}, t) \quad (1)$$

where \mathbf{u} is the particle displacement vector, $\boldsymbol{\sigma}$ is the stress tensor, ρ is density, \mathbf{b} corresponds to external body forces, and $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ in cartesian coordinates. Vectors and tensors are denoted with bold letters. scalars are Equation (1) is derived through the conservation of linear momentum. The propagation of elastic waves leads to longitudinal (P) waves, transversal (S) waves, P-to-S wave conversions, besides free surface phenomena such as Rayleigh and Love waves [3].

For a linear elastic non-dissipative medium, the relationship between stresses and strains $\boldsymbol{\varepsilon}$ is given by $\boldsymbol{\sigma} = \mathbf{C} : \boldsymbol{\varepsilon}$. The fourth order elastic tensor \mathbf{C} has between 2 up to 21 variables depending on the degree of anisotropy of the materials being considered [40]. The complexity of \mathbf{C} influences computational cost. For instance, an efficient

finite difference implementation of a wave propagator for a relatively large problem (768³ DoFs) considering transverse isotropic medium is about five times slower than compared to an isotropic medium [22].

An often adopted alternative [18] is to model the P-wave propagation using the acoustic wave equation, which is obtained by assuming an isotropic medium and neglecting shear strains:

$$\frac{1}{\kappa(\mathbf{x})} \frac{\partial^2 p}{\partial t^2}(\mathbf{x}, t) - \nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}, t) \right) = -\nabla \cdot \mathbf{b}(\mathbf{x}, t) \quad (2)$$

where κ is the bulk modulus relating scalar pressure p and displacement \mathbf{u} via the expression $p = -\kappa \nabla \cdot \mathbf{u}$. If the density varies significantly slower than the pressure field, Equation (2) can be simplified by making the assumption of constant density in the medium to:

$$\frac{\partial^2 p}{\partial t^2}(\mathbf{x}, t) - c^2(\mathbf{x}) \nabla^2 p(\mathbf{x}, t) = -\rho c^2(\mathbf{x}) \nabla \cdot \mathbf{b}(\mathbf{x}, t) \quad (3)$$

where $c = \sqrt{\kappa/\rho}$ is the wave speed.

Equations 2 and 3 are frequently used in active source seismic imaging [12]. Despite not representing the full complexity of the propagation of waves, the acoustic wave equation can still suffice. For one, not all data acquisition equipment can effectively capture or utilize more complex wave propagation physics. Secondly, solving a scalar partial differential equation (PDE) (3) is considerably computationally cheaper and requires less run-time memory than the vectorial PDE required by the elastic wave equation (1).

If the wave propagation constitutes a step of an imaging workflow, the number of distinct material parameters is also relevant to computational cost. For example, while the acoustic approximation for constant density (3) can be defined in terms of the wave speed c , the wave equation with varying density (2) needs the inversion of two independent fields: ρ density and P-wave velocity.

In this work the acoustic wave equation in its 2^nd order form with either constant (3) or variable density (2) was discretized using the finite difference method. Both the constant and variable density finite difference stencils' accuracy goes to up to 20th order in space and can be controlled at run time by the user. A second order central finite difference approximation is employed for the time derivative to create an explicit time-stepping scheme.

2.1 Boundary conditions and domain truncation

The application of boundary conditions and domain truncation techniques play an important role in the simulation of the acoustic wave. For example, applications such as non destructive testing and medical imaging workflows often need to enforce Dirichlet boundary conditions to emulate a free-surface. Seismic applications often need to damp simulated waves from reflecting off domain boundaries. In these cases, domain

truncation methods can be used to effectively absorb outgoing waves from the interior of a computational region without reflecting them back into the interior but at the cost of additional terms in the governing equations.

In most acoustic wave applications, a combination of an absorbing boundary condition [11] and a domain truncation technique like an absorbing boundary layer are used. Occasionally, a special treatment is also required to represent the free-surface boundary to model reflections [34]. These boundary condition techniques range from enforcing Robin boundary conditions [6, 17] to more complex approaches that involve modifying the acoustic wave equation and augmenting the physical domain [5].

`simwave` currently supports both Neumann and Dirichlet boundary conditions, which can be used on any number of the domain boundaries in addition to a user-configurable absorbing boundary layer (ABL) [13]. In the case of an ABL, the domain becomes $\Omega = \Omega_0 \cup \Omega_{ABL}$ where Ω_0 is the physical domain and Ω_{ABL} is the additional layer of user-defined width to absorb outgoing waves. In the case of the ABL, a non-zero damping term η is added to the original wave equation within Ω_{ABL} :

$$\frac{\partial^2 p}{\partial t^2}(\mathbf{x}, t) + 2\eta \frac{\partial p}{\partial t}(\mathbf{x}, t) - c^2(\mathbf{x}) \nabla^2 p(\mathbf{x}, t) = -\rho c^2(\mathbf{x}) \nabla \cdot \mathbf{b}(\mathbf{x}, t) \quad (4)$$

where η is zero everywhere except in the ABL. In the ABL, $\eta = \alpha d(\mathbf{x})^p$ in which α and p are two parameters that control the profile of the damping function, while $d(\mathbf{x})$ is the shortest distance from \mathbf{x} to the Ω_0 .

2.2 Time discretization

By multiplying Eq. (2) by the density ρ and expanding the expression under the divergent operator, the acoustic wave equation for variable density (with damping) at the instant $t = t_n$ may be written as:

$$\frac{1}{c^2} \left(\frac{\partial^2 p}{\partial t^2}(t_n) + 2\eta \frac{\partial p}{\partial t}(t_n) \right) + \frac{\nabla \rho}{\rho} \cdot \nabla p(t_n) - \nabla^2 p(t_n) = f(t_n) \quad (5)$$

The time axis is discretized uniformly such that $t_n = n\Delta t$ for $n = 0, \dots, N$ under a certain time step size Δt . A second-order accurate in time central finite difference scheme is chosen to approximate the time derivatives

$$\frac{dp}{dt}(t_n) \approx \frac{p^{n+1} - p^{n-1}}{2\Delta t}, \quad \frac{d^2 p}{dt^2}(t_n) \approx \frac{p^{n+1} - 2p^n + p^{n-1}}{\Delta t^2}. \quad (6)$$

In that case, an explicit time stepping scheme is obtained:

$$p^{n+1} = \frac{c^2 \Delta t^2}{1 + \eta \Delta t} (\nabla^2 p^n - \frac{\nabla \rho}{\rho} \cdot \nabla p^n) + 2p^n - (1 - \eta \Delta t) p^{n-1} \quad (7)$$

In practical applications, it remains important to be able to automatically determine a numerically stable timestep for the discretization. For the second-order

timestepping method used in this work, the necessary condition to select a numerically stable timestep Δt is given by [21]:

$$\Delta t \leq \frac{2 \Delta x}{c_{max} \sqrt{a}} \quad (8)$$

in which c_{max} is the maximum seismic velocity in the domain, Δt is the maximum timestep that can remain numerically stable, Δx is the grid spacing, and a is the sum of the finite difference coefficients involved with the spatial derivative terms in the wave equation. Note that a considers the usage of effect of arbitrarily higher order stencils for space derivative terms.

The timestepping scheme was implemented in a way such that wave propagators only need to keep in memory at most two time levels simultaneously, which reduces run-time memory load.

2.3 Space discretization

The computational domain is discretized with a regular grid with uniform spacing Δx_i in each axis x_i , where i goes from 1 up to 2 in 2D and 3 in 3D.

The spatial derivatives are approximated by central finite differences of even spatial orders up to 20. Along the x_i axis, the first and second derivatives at $x_i = k$ read as:

$$\partial_{x_i x_i} \phi_k = \frac{1}{\Delta x_i^2} (v_0 \phi_k + \sum_{j=1}^r v_i (\phi_{k+j} + \phi_{k-j})) \quad (9)$$

$$\partial_{x_i} \phi_k = \frac{1}{2\Delta x_i} (\sum_{j=1}^r w_i (\phi_{k+j} - \phi_{k-j})) \quad (10)$$

in which v_i are the coefficients of even spatial order for central finite difference schemes for second-order derivatives, w_i are the weights for central finite difference schemes for even spatial order for first-order derivatives, and r represents the stencil radius. The fully discretized stencil is obtained by substituting the expressions from (9) and (10) into Eq. (7). At the boundary, the domain is augmented with a number of ghost nodes that depends on the order of the stencils used to discretize the spatial derivatives.

2.4 Sources and receivers

The approach detailed in [16] is used to implement a body force at an arbitrary location within the grid and also to interpolate wave field solutions to receiver locations. Briefly, the source term is given by:

$$f_n = S d_n = S [W(n + \alpha) \text{sinc}(n + \alpha)] \quad (11)$$

in which $-0.5 < \alpha \leq 0.5$ and n represents an integer denoting a grid point, d_n represents a band-limited spatial delta function, and S is a time-varying wavelet.

The band-limited spatial delta function d_n is represented using a Kaiser window. The window function W given by:

$$W(x) = \begin{cases} \frac{I_0(b\sqrt{1-(x/r)^2})}{I_0(b)}, & \text{for } -r \leq x \leq r \\ 0, & \text{otherwise} \end{cases}$$

with the one free parameter b associated with the window, the half-width of the filter r , and I_0 is the zeroth-order Bessel function of the first kind. Optimal values for b from [16] are programmed for wavenumbers $k_{max} = \frac{1}{2}\pi$ given varying r . Ideally, the value of r should be kept as low as possible; however, this depends on the application and the desired numerical accuracy. With that said, the user can specify the desired value for r .

Figure 2 displays an example of a Kaiser Window W together with a *sinc* function and the corresponding weights multiplying the grid point values. The source (or receiver) is at a distance of 0.5 points from its neighbors, $b = 6.31$ and $r = 4$ in this instance.

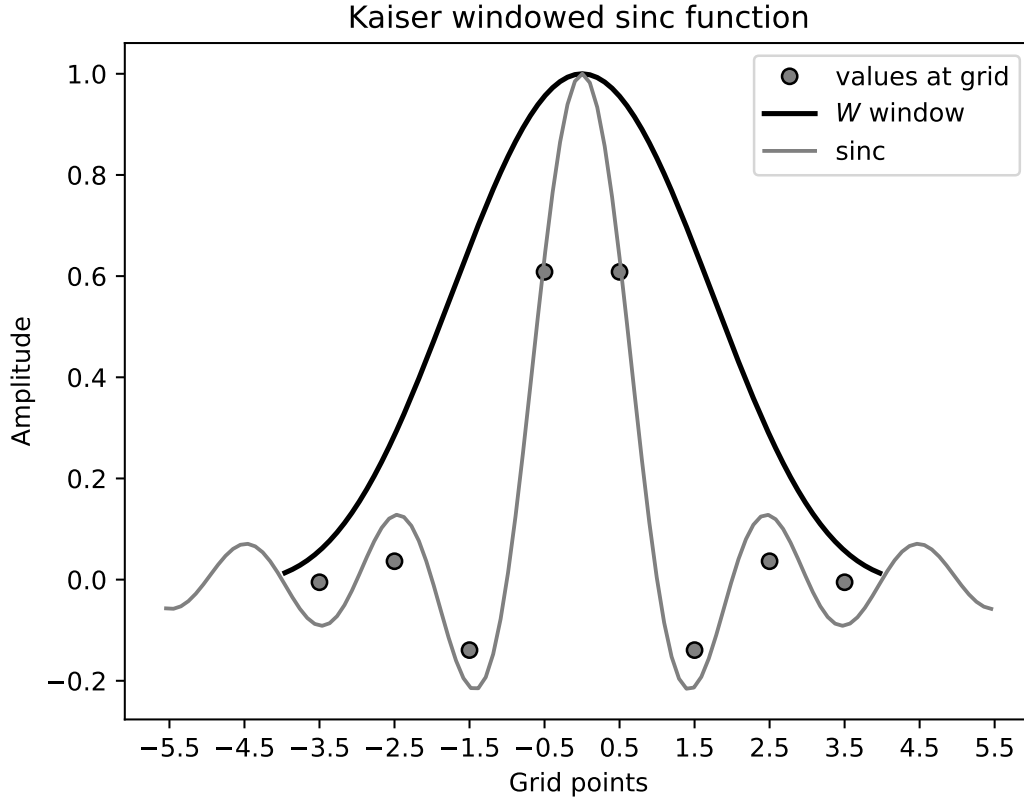


Figure 2: Windowing for a point at a distance of 0.5 grid points from its neighbors.

It similarly follows that the wave field solution p_n can be recorded to a set of

arbitrary receiver locations R in either 2D or 3D through:

$$R = \sum_{n=-r}^r p_n d_n \quad (12)$$

simwave permits the user to define an arbitrary time-varying wavelet S . By default, a function to generate a time-varying Ricker wavelet for a user-specified peak frequency is implemented.

2.5 Verification of numerical implementation

In order to verify that the numerical solutions produced by **simwave** are mathematically correct, we conduct several convergence tests in which we compare the order of accuracy of the discretized wave equation against theoretical values.

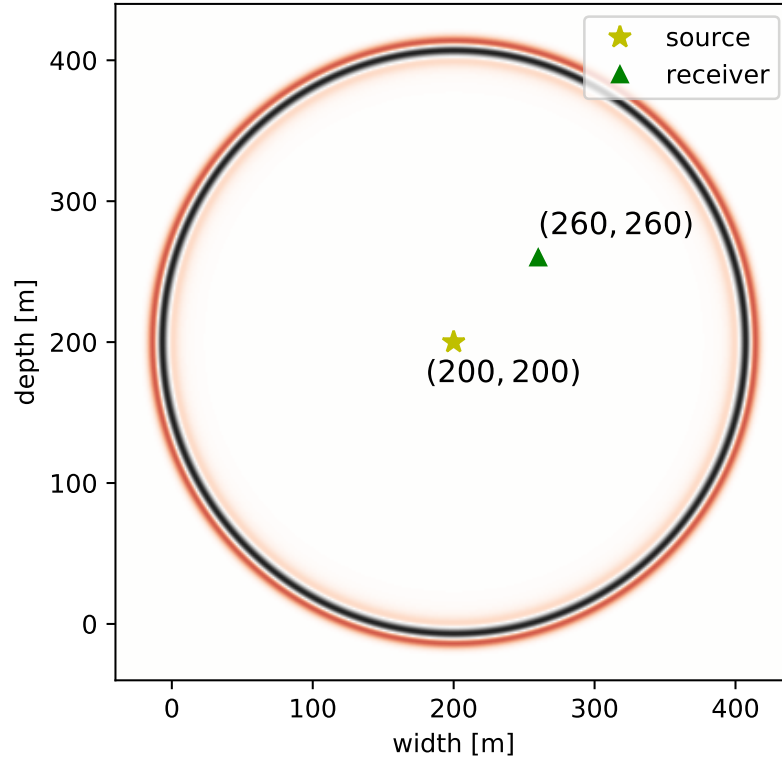


Figure 3: Simulation setup for the verification of the acoustic wave equation implementation.

A domain of 400×400 meters consisting of a homogeneous velocity model with $c = 1.5$ km/s is considered. At the center of the domain, a point source with a time

varying signal $s(t)$ produces a wavefield $u(\mathbf{r}, t)$, where \mathbf{r} denotes the distance from the source. A receiver at a distance of approximately 85 meters from the source registers the wave amplitude for $t = 150$ microseconds (Figure 3). The wave at the final instant $t = 150$ ms is also plotted, showing that the wave front never reaches the computational boundary. A Kaiser window width of 4 points is used both for source injection and receiver value interpolation. Wave and velocity field, as well the values collected at receivers are represented as single precision floating point numbers.

Numerical solutions are compared to an analytical solution [43] given by:

$$u(\mathbf{r}, t) = -\frac{i}{2} \int_{-\infty}^{\infty} H_0^{(2)}\left(\frac{\omega}{c}\mathbf{r}\right) \hat{s}(\omega) e^{i\omega t} d\omega \quad (13)$$

where $H_0^{(2)}$ is the Henkel function of second kind and \hat{s} is the Fourier transform of the original signal s . This analytical solution is valid as long as the source is punctual and boundary effects can be ignored.

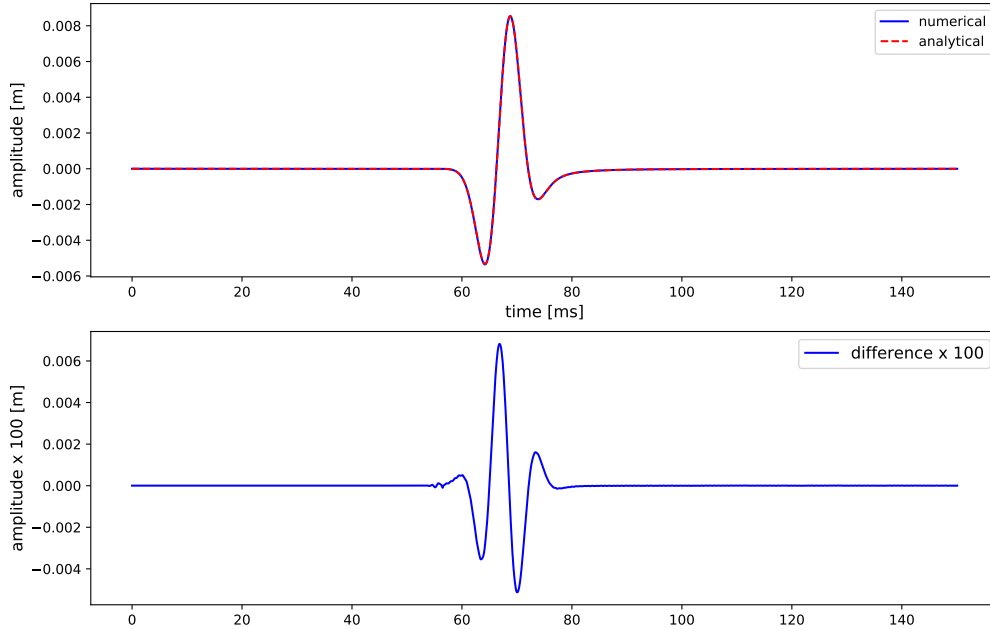


Figure 4: Comparison between numerical and analytical solution.

The domain is discretized as a square grid with spacing of 0.5 meters between nodes along both axes, and the time axis is discretized with a timestep of 0.1 ms. As shown in Fig. 4, the numerical solution is able to reasonably approximate the analytical one, as their difference is two orders of magnitude lower than the amplitudes at the receiver.

In order to verify the time discretization, we fix the spatial grid with spacing $h = 0.5$ meters and evaluate the Euclidean norm of the difference between the numerical solution u_{ref} and the exact solution u_{exa} at the receiver location. Since the time finite difference stencil employed is of second order, the error should decrease to the second order as $O(\Delta t^2)$. Fig. 5 displays the convergence rate alongside the theoretical curve, which demonstrates good agreement between theoretical and observed values.

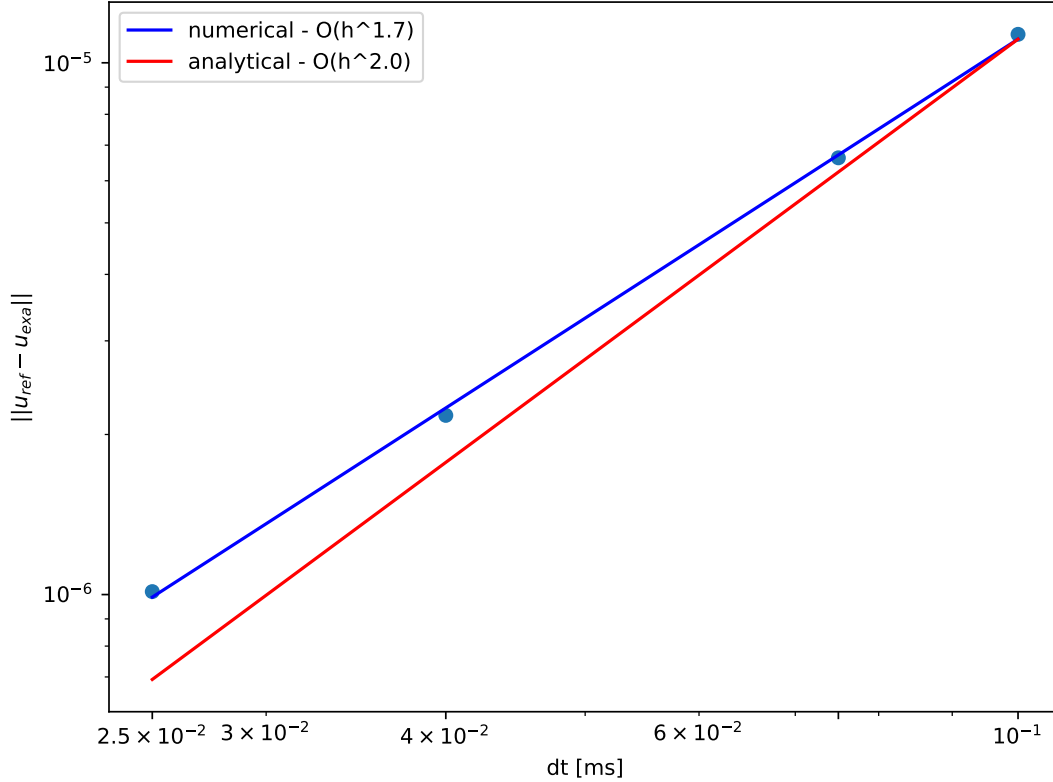


Figure 5: Numerical accuracy rate in time.

A similar analysis is performed regarding the space discretization. For a sufficiently small and fixed $dt = 0.025$ ms to minimize the influence of time discretization error, the spatial error is evaluated for different values of grid spacing h . Stencils with spacial orders up to 10 are considered and results are shown in Fig. 6. The convergence rates agree well with theoretical values for orders up to 8, and start to diverge from it as the magnitude of the spatial error becomes of the same magnitude as the time discretization error.

Finally, the wave equation with variable density is verified by the Method of Manufactured Solutions (MMS) [35]. A domain of 440×440 meters consisting of a homogeneous velocity model with $c = 2\text{km/s}$ is considered. A point source at the center of

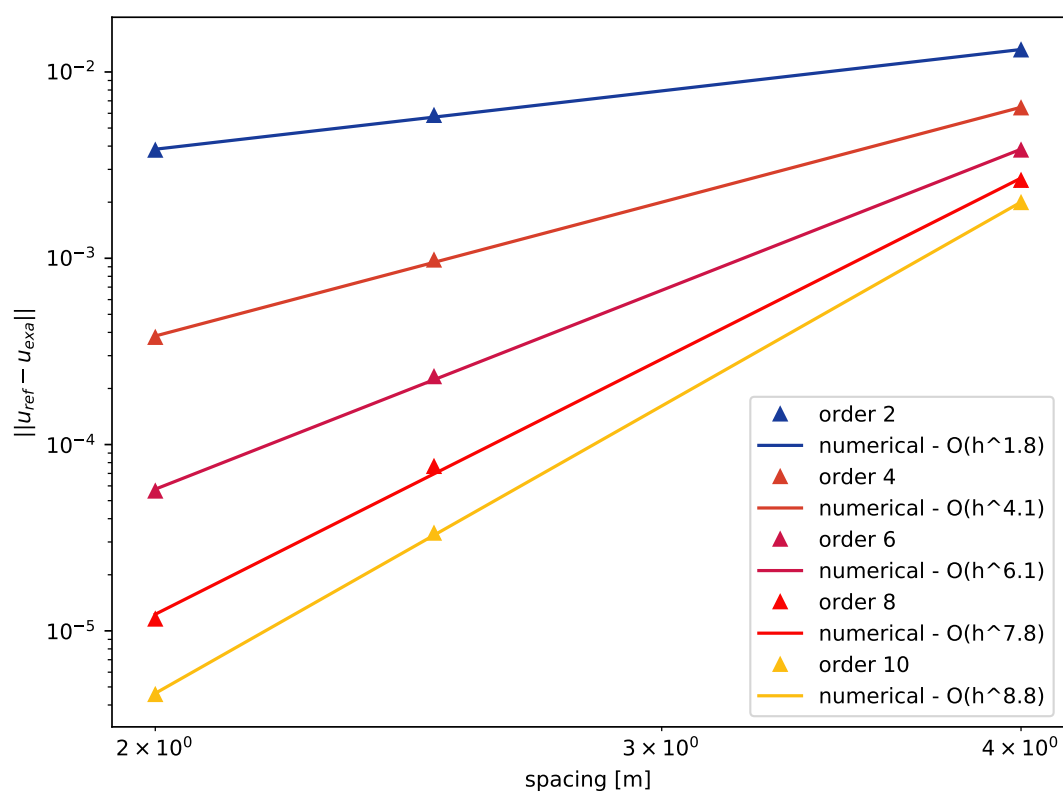


Figure 6: Numerical accuracy rate in space.

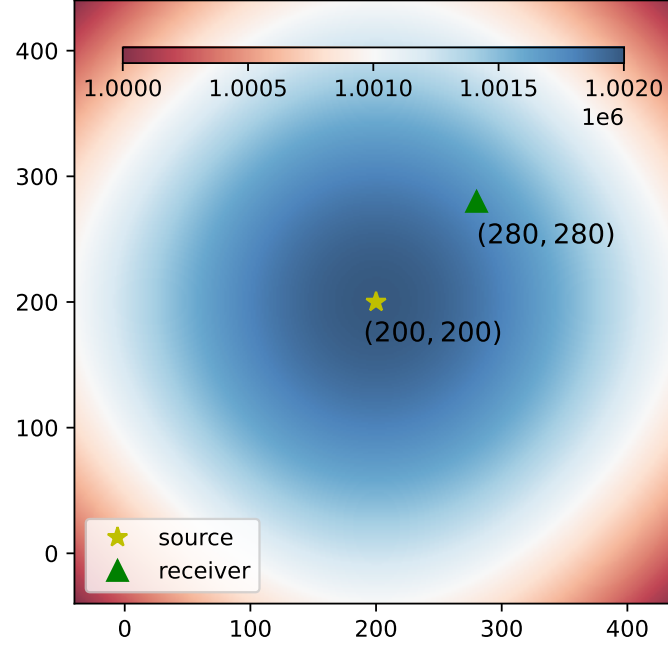


Figure 7: Density distribution used in the verification of the acoustic wave equation with variable density. The position of the source/receiver pair is also shown.

the domain produces a wavefield $u^*(x, z, t)$, where x, z are Cartesian coordinates. A receiver at a distance of approximately 113 meters from the source registers the wave amplitude for $t = 200$ microseconds as displayed in Figure 7. The time interval is discretized with a timestep of 0.05 ms. All relevant fields are represented with double precision floating point numbers. In order for the use of the variable density equation to be meaningful, a spatially varying density field ρ is chosen:

$$\rho = \left(1000 + \sin\left(\frac{\pi}{440}x\right) \right) \left(1000 + \sin\left(\frac{\pi}{440}z\right) \right) \quad (14)$$

The MMS consists in deriving the forcing term and boundary conditions for a PDE from a given solution. The following field is chosen as the ansatz:

$$u^*(x, z, t) = \sin\left(\frac{\pi}{440}x\right) \sin\left(\frac{\pi}{440}z\right) \sin(20\pi t) \sin(20\pi(t + dt)). \quad (15)$$

The density field ρ is plotted in Figure 7. The solution u^* has the same spatial dependency as the density, while the time dependency is shown in Figure 8. The appropriate forcing is derived by direct substitution into Eq. (2). One can also verify that u^* satisfies Dirichlet boundary conditions. The dependency in time is so that u^* is zero for

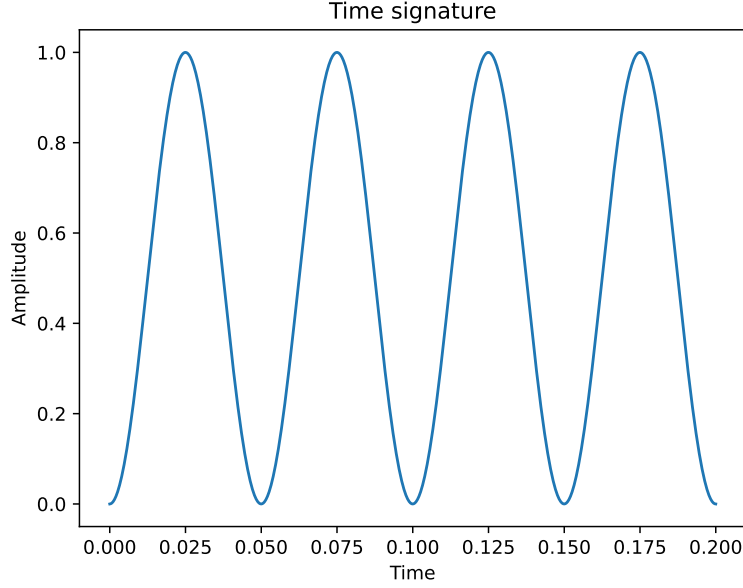


Figure 8: Time dependency of the solution u^* used in the verification of the acoustic wave equation with variable density.

the two first time steps $t = 0$ and $t = dt$. Figure 9 displays a comparison between analytical and numerical solutions for several grid spacing values. The numerical solution seems reasonably able to approximate the analytical solution, since as the grid spacing gets smaller, the numerical solution approaches the expected theoretical values.

3 Code architecture and implementation

For better separation of concerns, the architecture of `simwave` is organized into two layers (Figure 10). A Python front-end is implemented to provide a user-friendly interface which facilitates application development and integration with other scientific software libraries such as SciPy [41] and many others. A minimum body of knowledge is required from the application developer, for choosing a back-end, a compiler and its flags. All parallel processing strategies and hardware specific optimizations are implemented in the back-end. The performance critical components are implemented in the back-end which is written in ANSI C (sequential), or in C plus some support for parallelism (e.g., OpenMP, OpenACC, etc). The integration between the front-end and back-end uses Ctypes.

3.1 The front-end

The front-end provides the Python classes and functions with intuitive design for domain application programmers. The simulation of a wave propagation is performed by configuring and instantiating a `Solver` object. The solver aggregates a set of objects

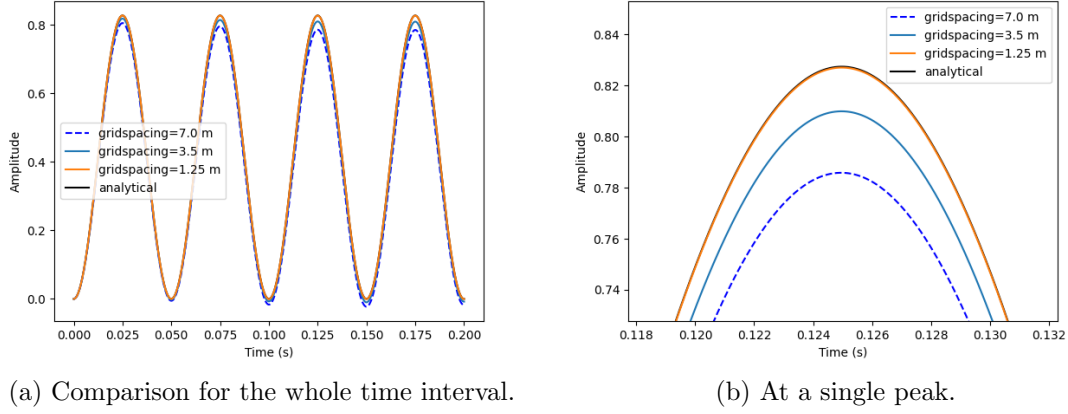


Figure 9: Comparison between numerical and analytical solution for variable density at the receiver location (280, 280).

that encapsulate important simulation parameters including:

- **SpaceModel**: This class defines the domain as a 2D or 3D axis-aligned regular Cartesian grid and requires additional numerical parameters to specify the spatial discretization. It configures the spatial order of the finite difference stencil. Boundary conditions and absorbing layers are also enabled by calling its method `config_boundary()`. The **SpaceModel** class requires grid-shaped dataset containing scalar values for all the grid points. For example, in seismology the seismic velocity values are typically supplied, while the spatially variable density is optional.
- **TimeModel**: Objects of this class encapsulate temporal discretization parameters for the wave simulation, such as the start time, end time, and the timestep. This class can automatically calculate a numerically stable simulation timestep Δt that respects the CFL conditions [21] from a **SpaceModel** object. The user can optionally specify Δt if needed.
- **Source**: This class implements source injection as described in Section 2.4 according to the quantity and their locations in the domain provided by the programmer. Notice that multiple sources can be enforced simultaneously.
- **Receiver**: similar to the **Source** class, the **Receiver** represents a set of receivers positions across the domain. These receivers represent recording devices (e.g. hydrophones) that record wave signals and can be used to generate seismograms for the simulation.
- **Wavelet**: This class represents a time varying wavelet to be injected into the domain. The user can specify a custom call-back function that describes the

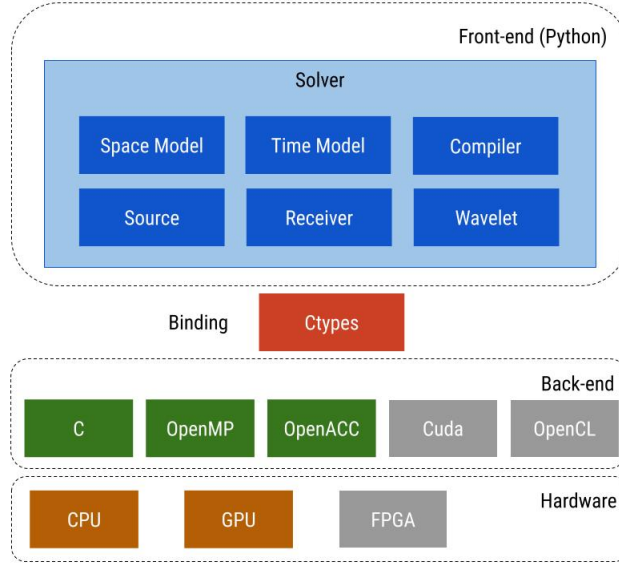


Figure 10: The architecture of `simwave`: the components in gray will be implemented in future work.

variation in time of the body force. `simwave` also provides a `RickerWavelet` default sub-class which extends the `Wavelet` and implements a Ricker wavelet.

- **Compiler:** This object encapsulates compilation parameters for the generation of C code, such as the compiler implementation (e.g. gcc, icc, clang) and compiler flags. The `Compiler` class is responsible for compiling the C code and generating a shared object at run time. Despite belonging to the front-end stack, this object is used to generate the back-end code.

3.2 The back-end

The back-end layer, which solves the PDEs and simulate the propagation of acoustic waves, is implemented in C programming language in a compact and modular design to facilitate its parallelization and optimization for modern HPC hardware. The back-end kernel implements stencil codes [9] which are compiled and linked according to the hardware specified by the application programmer. Parameters provided by the front-end guide the generation of the back-end, which can implement either serial (baseline) or parallel code (in OpenMP or OpenACC), for 2D or 3D domains, to solve the acoustic wave with constant or variable density (Equations 2 and 3, respectively), to execute on CPUs or GPUs. Once the back-end code is generated, it receives data structures initialized in the front-end and passed by parameters through `Ctypes`. The back-end executes the simulation and returns final results to the front-end.

The back-end supports all the concerns related to parallelism, performance, hardware specific optimizations and performance portability. Besides providing a reference

implementation which is numerically correct, the baseline (serial) code can also be used as an industry proxy of seismic applications for research in high performance computing (HPC) [27, 31, 32, 33, 45]. In its first release, **simwave** implements three back-ends: sequential C (baseline), OpenMP, and OpenACC. The two later can generate code for CPUs of different architectures (e.g., x86, ARM, AMD, Power) and for GPUs. In the future, novel back-ends may be developed using technologies like OpenCL, DPC++, CUDA, and others.

4 Example of use

This section illustrates the use of **simwave** for the simulation of two examples. Listing 1 shows the use of the **simwave** to simulate acoustic waves propagation with the Marmousi2 P-wave velocity model (Figure 11) [25] in a two dimensional domain which has 3.5 km depth by 17 km width. Other external packages (e.g. **scipy**, **matplotlib**, **numpy**) can be used together for data visualization.

```

1 from simwave import (
2     SpaceModel, TimeModel, RickerWavelet, Solver, Compiler,
3     Receiver, Source, read_2D_segy,
4     plot_wavefield, plot_shotrecord, plot_velocity_model
5 )
6 import numpy as np
7
8 # Marmousi2 velocity model
9 marmousi_model = read_2D_segy('MODEL_P-WAVE_VELOCITY_1.25m.segy')
10
11 compiler = Compiler(
12     cc='gcc',
13     language='cpu_openmp',
14     cflags='-O3 -fPIC -ffast-math -std=c99'
15 )
16
17 # create the space model
18 space_model = SpaceModel(
19     bounding_box=(0, 3500, 0, 17000),
20     grid_spacing=(10.0, 10.0),
21     velocity_model=marmousi_model,
22     space_order=4,
23     dtype=np.float64
24 )
25
26 # config boundary conditions
27 space_model.config_boundary(
28     damping_length=(0, 700, 700, 700),
29     boundary_condition=(
30         "null_neumann", "null_dirichlet",
31         "null_dirichlet", "null_dirichlet"
32     ),
33     damping_polynomial_degree=3,
34     damping_alpha=0.001
35 )

```



```

36
37 # create the time model
38 time_model = TimeModel(
39     space_model=space_model,
40     tf=2.0
41 )
42
43 # create the set of sources
44 source = Source(
45     space_model,
46     coordinates=[(20, 8500)],
47     window_radius=1
48 )
49
50 # create the set of receivers
51 receiver = Receiver(
52     space_model=space_model,
53     coordinates=[(20, i) for i in range(0, 17000, 10)],
54     window_radius=1
55 )
56
57 # create a ricker wavelet with 10hz of peak frequency
58 ricker = RickerWavelet(10.0, time_model)
59
60 # create the solver
61 solver = Solver(
62     space_model=space_model,
63     time_model=time_model,
64     sources=source,
65     receivers=receiver,
66     wavelet=ricker,
67     saving_stride=0,
68     compiler=compiler
69 )
70
71 # run the forward
72 u_full, recv = solver.forward()
73
74 # remove damping extension from u_full
75 u_full = space_model.remove_nbl(u_full)
76
77 extent = [0, 17000, 3500, 0]
78
79 # plot the velocity model
80 plot_velocity_model(space_model.velocity_model, extent=extent)
81
82 # plot the last wavefield
83 plot_wavefield(u_full[-1], extent=extent)
84
85 # plot the seismogram

```

86 `plot_shotrecord(recv)`

Listing 1: Forward simulation in a two dimensional domain using Marmousi2 velocity model.

After reading the velocity model (in line 9), we define the compiler options (lines 11-15) by instantiating an object `Compiler`. This object defines a set of compiler choices and flags including the C compiler, the compilation flags, the `language` which enables sequential or parallel implementation (in OpenMP or OpenACC), and the target architecture (i.e., CPU or GPU). Optionally, it is possible to override the baseline code by pointing out to the path to a custom C implementation as kernel through the parameter `cfile`. This can be useful to evaluate new strategies and HPC techniques.

Following this, we configure the spatial domain with the object `SpaceModel` (line 18). The `bounding_box` attribute defines the domain boundaries (the begin and the end) in meters along the axis, respectively Z (depth) and X (width). The `grid_spacing` defines the spacing (in meters) between grid points for each axis of the domain. The total grid size is calculated according to domain size (`bounding_box`) and the `grid_spacing`. The `space_order` defines the finite differences spatial order, which can be any even order ranging from 2 to 20. The `dtype` sets the numeric precision, e.g., `numpy.float32` for single-precision, and `numpy.float64` for double-precision. The velocity model is represented as a numpy array in either two or three dimensions and expressed in meters per second by the `velocity_model` parameter. The optional attribute `density_model` specifies the density of materials in each grid point. When the density is provided, the acoustic equation with variable density (Equation 3) is automatically used for the simulation. In this case, the density model is also represented as a numpy array and carries the units of g/cm^3 . Both the velocity and density models are linearly interpolated to fit the domain extent.

To enforce boundary conditions, line 27 invokes the method `config_boundary` of `SpaceModel`. The `damping_length` parameter defines the domain extension length (in meters) for the damping on each border of the domain, respectively Z (top and bottom) and X (left and right) in the 2D domain, and Y (front and back) in a 3D case. The parameter `boundary_condition` defines the boundary condition applied on each side of the domain. The options include `null_neumann`, `null_dirichlet`, and `none`. The parameters `damping_polynomial_degree` and `damping_alpha` are referred to as p and α in Equation 4.

Next, (in line 38) we configure the time model by instantiating an object of the class `TimeModel`, providing the object `space_model` which contains spatial information (e.g. space order, domain dimension, maximum p-wave velocity) required to calculate the critical Δt . This object also encapsulates `tf`, which defines total propagation time (in seconds) for the simulation, and `saving_stride` which sets the wave field saving configuration. The saving stride can be zero (only the snapshot in the last time step is returned), one (the snapshots of all time steps are returned) or any number n ($1 < n < \lceil \frac{tf}{\Delta t} \rceil$) which determines saving every n time steps (i.e., the stride). Optionally, the user can define a custom Δt through the optional parameter `dt`, otherwise the

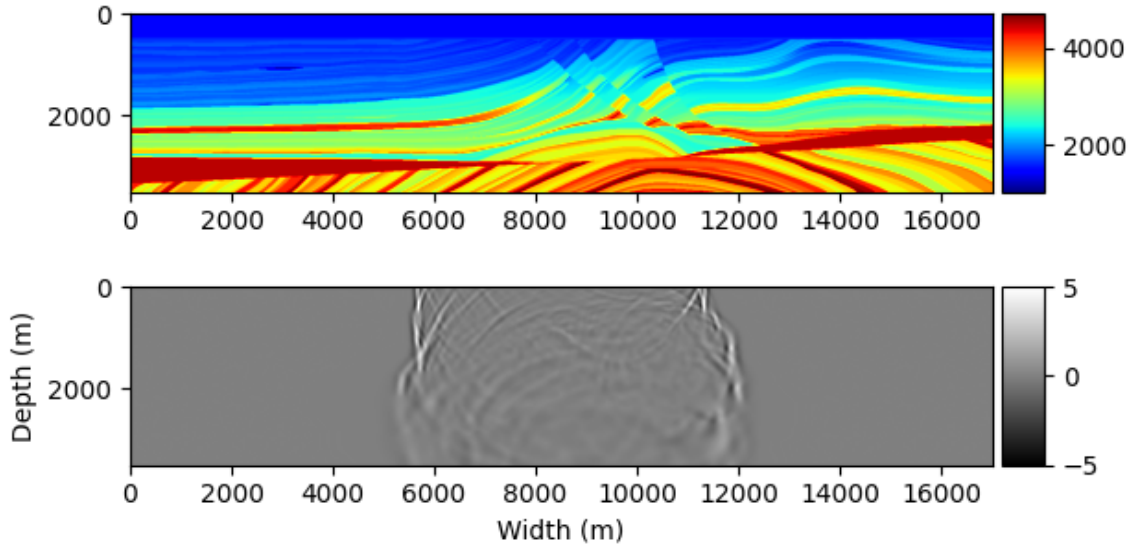


Figure 11: Marmousi2 velocity model (top) and final wavefield of the forward simulation (bottom).

default critical Δt is applied.

Next, we define the sources (in line 44) and receivers (line 51) by providing the grid (the `space_model` object), the `coordinates`, and `window_radius`. The parameter `coordinates` is a list of tuples containing the coordinates of sources or receiver in meters in the domain. The `window_radius` defines the radius (ranging from 1 to 10) of the Kaiser window applied in source/receiver interpolation. In the example, a Ricker wavelet is applied with a peak frequency of 10 Hertz. Notice that the wavelet requires the `TimeModel`.

An object `Solver` is instantiated by aggregating all the previous objects that configure the simulation. The method `forward` executes the simulation, returning the full wave field and the seismogram after conclusion. The Figure 11 shows the Marmousi2 velocity model (top) and the final wave field of the simulation (bottom), while Figure 12a depicts the corresponding seismogram.

The next example (in Listing 2) shows the use of `simwave` to simulate the propagation of an acoustic wave on the Overthrust velocity model [2] in a three dimensional domain. The Overthrust model (depicted in Fig. 13) has 4.12 km in depth, 16 km in width and 16 km in length. The source code is very similar to the previous 2D example with the addition of one dimension. The final wave field produced by this simulation is shown in Fig. 14, and the seismogram is shown in Fig. 12b.

```

1 from simwave import (
2     SpaceModel, TimeModel, RickerWavelet, Solver,
3     Compiler, Receiver, Source
4 )
5 import numpy as np

```

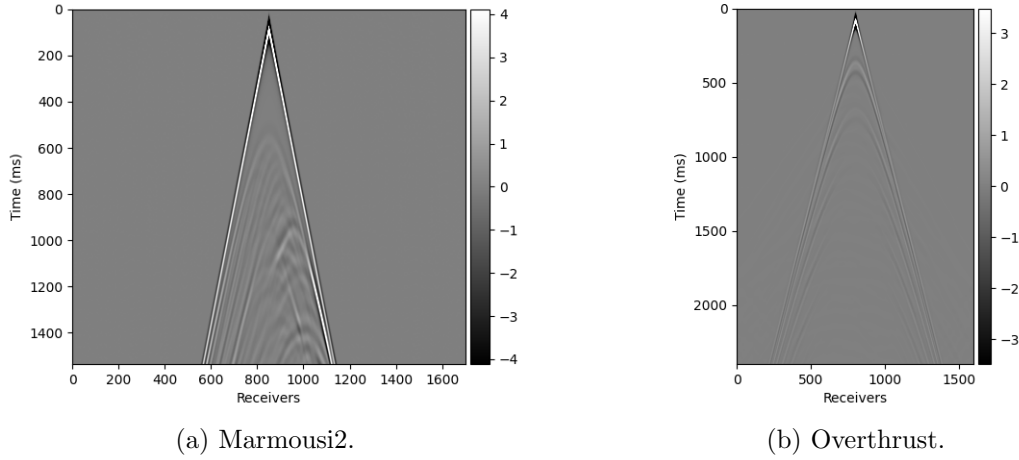


Figure 12: Seismogram from the the forward simulation.

```

6 import h5py
7
8 def read_model(filename):
9     with h5py.File(filename, "r") as f:
10
11         # Get the data
12         data = list(f['m'])
13
14         data = np.array(data)
15
16         # convert to m/s
17         data = (1 / (data ** (1 / 2))) * 1000.0
18
19         return data
20
21 if __name__ == '__main__':
22
23     data = read_model('overthrust_3D_true_model.h5')
24
25     compiler = Compiler(
26         cc='clang',
27         language='gpu_openmp',
28         cflags='-O3 -fPIC -ffast-math -fopenmp \
29             -fopenmp-targets=nvptx64 -Xopenmp-target'
30     )
31
32     space_model = SpaceModel(
33         bounding_box=(0, 4120, 0, 16000, 0, 16000),
34         grid_spacing=(20., 20., 20.),
35         velocity_model=data,
36         space_order=4,

```

```

37     dtype=np.float64
38 )
39
40 space_model.config_boundary(
41     damping_length=0,
42     boundary_condition=(
43         "null_neumann", "null_dirichlet",
44         "null_dirichlet", "null_dirichlet",
45         "null_dirichlet", "null_dirichlet"
46     ),
47     damping_polynomial_degree=3,
48     damping_alpha=0.001
49 )
50
51 time_model = TimeModel(
52     space_model=space_model,
53     tf=4,
54     saving_stride=0
55 )
56
57 source = Source(
58     space_model,
59     coordinates=[(20, 8000, 8000)],
60     window_radius=1
61 )
62
63 receiver = Receiver(
64     space_model=space_model,
65     coordinates=[(40, 8000, i) for i in range(0, 16000, 10)],
66     window_radius=1
67 )
68
69 ricker = RickerWavelet(8.0, time_model)
70
71 solver = Solver(
72     space_model=space_model,
73     time_model=time_model,
74     sources=source,
75     receivers=receiver,
76     wavelet=ricker,
77     compiler=compiler
78 )
79
80 # run the forward
81 u_full, recv = solver.forward()

```

Listing 2: Forward simulation in a three dimensional domain using the Overthrust velocity model.

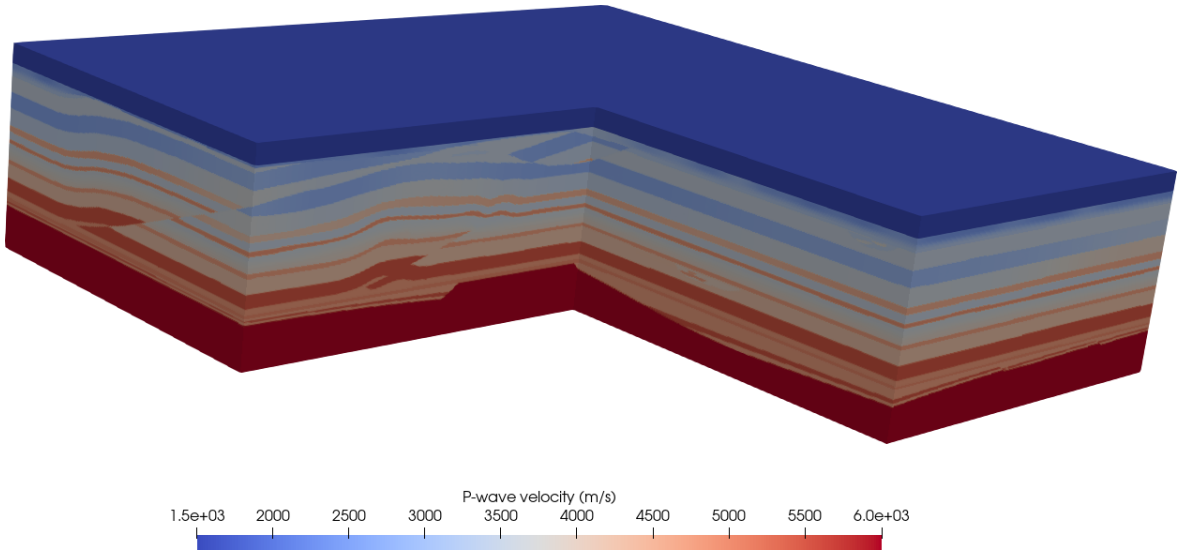


Figure 13: The Overthrust P-wave velocity model.

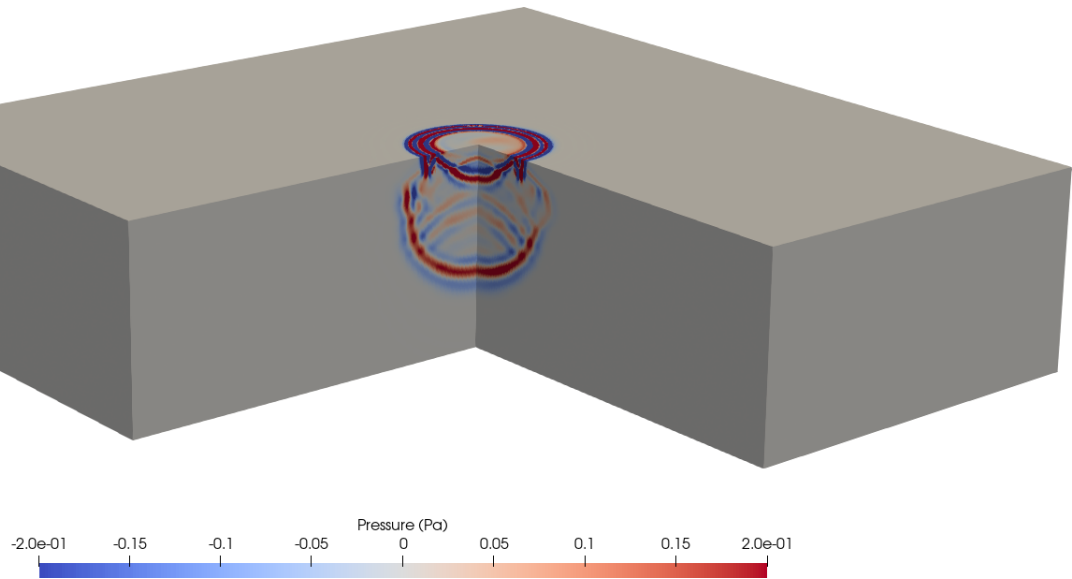


Figure 14: Wave field at $t = 1.0$ s for the Overthrust benchmark.

5 Performance evaluation

As wave simulation is the kernel of many large inversion problems, optimizing its performance for efficient execution on several HPC systems is mandatory. The two previous examples are used to assess performance on CPU and GPU systems. For the 2D performance test, a 2 second acoustic wave propagation using the Marmousi2 (Listing 1) benchmark in which a 3.5 km deep per 17 km wide domain is discretized with a 2D grid with 351 x 1701 points. A damping length of 700 m is added on each side except along the top boundary and results in a 421 x 1841 grid (775,061 grid points). The number of timesteps varies according to the spatial order, being 1331 for 2^{nd} , 1537 for 4^{th} , and 1696 for 8^{th} . For the 3D performance experiment, we simulated 4 seconds of wave propagation using the Overthrust 3D velocity model (Listing 2) with 4.12 km depth x 16 km width x 16 km length, discretized in a 3D grid with 207 x 801 x 801 points (132,811,407 grid points). Likewise, the number of timesteps varies in 2080 for 2^{nd} , 2401 for 4^{th} and 2651 for 8^{th} spatial order.

Benchmarks were executed in both CPU and GPU environments. The CPU execution was carried out in a cluster node with two Intel Xeon Gold 6148 processors (Skylake) with 20 cores each and 192GB of memory. The GPU executions were performed in the GeForce RTX 2080 Super (Turing architecture) and Tesla V100 (Volta architecture). Each benchmark was compiled with GCC 8.3 (GNU Compiler) in the CPU environment. For the GPU execution we used the PGCC 21.11 (PGI compiler) for offloading using OpenACC and CLANG 13.0 (LLVM project) for OpenMP. The flags applied in each compiler are listed in Table 1.

Table 1: Compiler flags used in the executions.

Compiler	Flags
GCC	-O3 -fPIC -ffast-math -std=c99
CLANG	-O3 -fPIC -ffast-math -fopenmp -fopenmp-targets=nvptx64 -Xopenmp-target
PGCC	-O3 -fPIC -acc:gpu -gpu=pinned

The experiment measured the execution time for both 2D and 3D acoustic wave propagator with constant density, discretized with 2^{nd} , 4^{th} , and 8^{th} spatial orders. For the CPU experiments we increased the number of cores from 1 core in the sequential version up to 40 cores available in the compute node. The execution on GPUs used all the available cores.

The simulations were repeated 10 times and the average execution times and speedups are presented in the Tables 2 (Marmousi 2D) and 3 (Overthrust 3D). Note that the speedup is calculated as the ratio of the parallel execution time to the serial execution time. Because such finite difference stencils are intrinsically memory-bound codes, the scalability in CPU is hindered when the number of cores is increased above the number of memory channels available (6 channels for this CPU). This result is consistent with other studies in literature (e.g., in [28, 30]).

Table 2: Execution times in seconds and speedup (parallel time / serial time) for the simulation of 2D Marmousi forward propagation. The best results are highlighted in bold.

Hardware	Back-end	Compiler	SO=2		SO=4		SO=8	
			Time	S	Time	S	Time	S
6148 - 1 core	C	gcc	4.18	1.0	6.27	1.0	10.03	1.0
6148 - 2 cores	OpenMP	gcc	2.31	1.8	3.40	1.8	5.38	1.9
6148 - 4 cores	OpenMP	gcc	1.27	3.3	1.87	3.4	2.90	3.5
6148 - 8 cores	OpenMP	gcc	0.71	5.9	1.04	6.0	1.57	6.4
6148 - 20 cores	OpenMP	gcc	0.39	10.7	0.57	11.1	0.83	12.0
6148 - 40 cores	OpenMP	gcc	0.31	13.3	0.48	13.1	0.69	14.5
RTX 2080 Super	OpenMP	clang	0.76	5.5	0.86	7.3	0.98	10.3
RTX 2080 Super	OpenACC	pgcc	0.61	6.9	0.66	9.5	0.77	13.0
V100	OpenMP	clang	0.36	11.5	0.43	14.6	0.47	21.3
V100	OpenACC	pgcc	0.41	10.1	0.45	14.1	0.49	20.5

For the 2D benchmark with spatial order 2, the CPU with 40 cores produces the best performance. However, for higher spatial orders, the GPU performs better. In the case of the GPU, there is a data transfer cost (CPU memory to GPU memory), but because the GPU has far higher throughput in terms of processing than the CPU, our results suggest larger workloads can be processed on the GPU more quickly. Further, the numerical solution of the wave equation implements stencil patterns which is memory-intensive, a scalability limiting factor [28, 36]. Thus, the memory bandwidth represents a bottleneck for performance and scalability.

Notice that 3D benchmark showed speedups significantly higher than the 2D because the 3D produces a significantly larger amount of work to execute. By calculation 132,811,407 grid points per time step, the 3D launches a massive number of work units (i.e., thread blocks) which can be executed in parallel as soon as their data arrive from the memory. This allows better hiding the memory latency of the GPU than the 2D benchmark which computes far less (775,061) grid points per time step.

Currently, **simwave** applies straightforward loop parallelism strategies supported by thread-based OpenMP or OpenACC compilers, and compiler-specific automatic optimizations (i.e. -O3). The investigation on more advanced loop optimization strategies is beyond of this work’s scope and will be addressed in future work.

6 Comparison with other simulation packages

simwave implements an explicit solver to simulate the propagation of acoustic waves with constant or variable density, based on the finite-difference method. There are plenty of software technologies used for Geophysics research, including those developed by communities [8, 42], companies, or individuals [7]. However, most software packages maintained by the CIG project [8] are not directly comparable to **simwave** because they

Table 3: Execution times (in seconds) and speedup for the simulation of 3D Overthrust forward propagation. The best results are highlighted in bold.

Hardware	Back-end	Compiler	SO=2		SO=4		SO=8	
			Time	S	Time	S	Time	S
6148 - 1 core	C	gcc	1642.41	1.0	2565.88	1.0	3909.55	1.0
6148 - 2 cores	OpenMP	gcc	901.51	1.8	1360.49	1.9	2048.54	1.9
6148 - 4 cores	OpenMP	gcc	475.52	3.5	716.31	3.6	1081.00	3.6
6148 - 8 cores	OpenMP	gcc	248.84	6.6	374.98	6.8	569.71	6.9
6148 - 20 cores	OpenMP	gcc	186.98	8.8	272.56	9.4	429.54	9.1
6148 - 40 cores	OpenMP	gcc	110.72	14.8	171.11	15.0	347.09	11.3
RTX 2080 Super	OpenMP	clang	72.46	22.7	93.95	27.3	130.23	30.0
RTX 2080 Super	OpenACC	pgcc	48.02	34.2	67.68	37.9	103.95	37.6
V100	OpenMP	clang	28.30	58.0	40.36	63.6	63.12	61.9
V100	OpenACC	pgcc	37.13	44.2	50.10	51.2	68.13	57.4

were designed with focus on specific aspects of earthquakes.

A comprehensive list in [7] compares dozens of software packages which focus on exploration geophysics. Likely the most widely known software for geophysics research, Madagascar [23] is designed for multidimensional data analysis and reproducible computational experiments which is distributed as an open-source package [1]. The objective is to provide an environment for researchers working with digital image and data processing in geophysics and related fields. The package consists of two levels: low-level main programs (typically developed in the C programming language and working as data filters) and high-level processing flows (described with the help of the Python programming language) that combine main programs and completely document data processing histories for testing and reproducibility. The package is composed of more than 1,000 programs that support a significantly broader range of functionalities if compared to **simwave**. Furthermore, Madagascar’s focus is to serve as a tool for reproducible research in several areas of geophysics, while **simwave** focus on simulate the propagation of acoustic waves.

A more closely related project is Minimod [26], which implements several solvers for the acoustic wave with constant density, acoustic wave with variable density, acoustic transversely isotropic, and the elastic equation. Minimod can serve both for geophysics research, and for HPC research as in [27, 31, 32, 33, 45]. However, Minimod is currently not publicly available by the time of this writing.

7 Quality control

Quality control is enforced with the support of **pytest** (<https://docs.pytest.org/>), and continuous integration and continuous delivery (CI/CD) mechanisms supported by GitHub. This enables automating the tests and running the software development workflows directly in the repository using the GitHub’s servers. Tests are executed on every push or pull requests to the master branch of the repository. Similarly, one

workflow uploads and updates the `simwave`'s package version in the PyPI every time a release is created. The test suite consists of functional and unit tests. The unit tests are important to check isolated pieces of the code, ensuring the expected outputs according to the inputs. And the functional tests are applied to verify slices of the application as well as the entire program. Part of the tests are black box, comparing the simulation output to known reference values. These values are obtained from problems that have analytical solutions (described in the validation section) and also from the output of earlier versions of `simwave`, which is a form of regression testing. Installation and testing instructions can be found in the `simwave`'s repository on GitHub, along with use case examples.

8 Availability

Operating system

`simwave` can be installed via `pip` package manager either from the source or from the Python Package Index (PyPI) on GNU/Linux, Mac OS X and on any platform supported by Docker, like Azure and AWS.

Programming language

Python 3.6 or newer and C.

Additional system requirements

Memory depending on domain size and use case.

Dependencies

The required `simwave` dependencies are listed below.

1. `numpy` $\geq 1.18.1$
2. `matplotlib` $\geq 3.2.1$
3. `segio` $\geq 1.9.1$
4. `scipy` $\geq 1.4.1$
5. `pytest` $\geq 6.2.2$
6. `pytest-codeblocks` $\geq 0.10.4$
7. `findiff` $\geq 0.8.9$

List of contributors

The development of **simwave** has the following contributors.

1. Jaime Freire de Souza (Federal University of Sao Carlos)
2. Keith Jared Roberts (University of Sao Paulo)
3. João Baptista Dias Moreira (University of Sao Paulo)
4. Roussian di Ramos Alves Gaiosio (Federal University of Sao Carlos)
5. Hermes Senger (Federal University of Sao Carlos)

Software location:

Archive

Name: Zenodo

Persistent identifier: <https://doi.org/10.5281/zenodo.5847017>

Licence: GNU General Public License v3.0

Publisher: Hermes Senger

Version published: v1.0

Date published: 13/01/2022

Code repository

Name: GitHub

Persistent identifier: <https://github.com/HPCSys-Lab/simwave>

Licence: GNU General Public License v3.0

Date published: 13/01/2022

Language

English.

Reuse potential

simwave can be used to simulate the propagation of acoustic waves in single- or multi-material domains with constant and variable density, such as in full-waveform inversion (FWI) [40] or reverse-time migration (RTM) [4, 39] problems. The simulations are written in Python and use **simwave** as a library to be imported and used either alone, or in combination with scientific libraries such as SciPy and others. The **simwave**'s code is provided in two forms, a sequential (baseline) and an accelerated implementation for users who need to cope with large problems. The code may also be used as a representative of relevant industrial codes which can serve as benchmark for research on high-performance computing methods, such as in [27, 31, 32, 33, 45]. Finally, users and researchers can get in touch with the development team through the **simwave**'s issue page on GitHub (<https://github.com/HPCSys-Lab/simwave/issues>).

Acknowledgements

The authors thank the support from Shell Brasil and ANP. The sixth author thanks the financial support of CNPq, Brazil under grant 302658/2018-1. The seventh author thanks the support of São Paulo Research Foundation (FAPESP), under grant 2019/26702-8.

Funding statement

The authors gratefully acknowledge sponsorship from Shell Brasil through the *ANP 20714-2 - Desenvolvimento de técnicas numéricas e software para problemas de inversão com aplicações em processamento sísmico* project at Universidade de São Paulo and the strategic importance of the support given by ANP through the R&D levy regulation.

Competing interests

The authors have no competing interests to declare.

References

- [1] ahay.org [2021], ‘Madagascar project’, <https://github.com/ahay/src>.
- [2] Aminzadeh, F. [1997], *3-D salt and overthrust models*, Society of Exploration Geophysicists, Tulsa, Oklahoma.
- [3] Atkin, R. J. and Fox, N. [2005], *An introduction to the theory of elasticity*, Courier Corporation.
- [4] Baysal, E., Kosloff, D. D. and Sherwood, J. W. [1983], ‘Reverse time migration’, *Geophysics* **48**(11), 1514–1524.
- [5] Berenger, J.-P. [1994], ‘A perfectly matched layer for the absorption of electromagnetic waves’, *Journal of computational physics* **114**(2), 185–200.
- [6] Clayton, R. and Engquist, B. [1977], ‘Absorbing boundary conditions for acoustic and elastic wave equations’, *Bulletin of the seismological society of America* **67**(6), 1529–1540.
- [7] *Comparison of free geophysics software* [2021], Wikimedia Foundation, Inc.
URL: https://en.wikipedia.org/wiki/Comparison_of_free_geophysics_software
- [8] *Computational Infrastructure for Geodynamics (CIG). Community website.* [n.d.], <https://geodynamics.org/cig/software/>. Accessed: 2021-10-04.
- [9] Datta, K., Kamil, S., Williams, S., Olikar, L., Shalf, J. and Yelick, K. [2009], ‘Optimization and performance modeling of stencil computations on modern microprocessors’, *SIAM Rev.* **51**(1), 129–159.
URL: <https://doi.org/10.1137/070693199>

- [10] Dushaw, B. D., Egbert, G. D., Worcester, P. F., Cornuelle, B. D., Howe, B. M. and Metzger, K. [1997], ‘A topex/poseidon global tidal model (tpxo. 2) and barotropic tidal currents determined from long-range acoustic transmissions’, *Progress in Oceanography* **40**(1-4), 337–367.
- [11] Engquist, B. and Majda, A. [1977], ‘Absorbing boundary conditions for numerical simulation of waves’, *Proceedings of the National Academy of Sciences* **74**(5), 1765–1766.
- [12] Fichtner, A. [2010], *Full seismic waveform modelling and inversion*, Springer Science & Business Media.
- [13] Gao, Y., Song, H., Zhang, J. and Yao, Z. [2017], ‘Comparison of artificial absorbing boundaries for acoustic wave equation modelling’, *Exploration Geophysics* **48**(1), 76–93.
- [14] Guasch, L., Agudo, O. C., Tang, M.-X., Nachev, P. and Warner, M. [2020], ‘Full-waveform inversion imaging of the human brain’, *NPJ digital medicine* **3**(1), 1–12.
- [15] Guasch, L., Calderón Agudo, O., Tang, M.-X., Nachev, P. and Warner, M. [2020], ‘Full-waveform inversion imaging of the human brain’, *npj Digital Medicine* **3**(1), 28.
URL: <https://doi.org/10.1038/s41746-020-0240-8>
- [16] Hicks, G. J. [2002], ‘Arbitrary source and receiver positioning in finite-difference schemes using Kaiser windowed sinc functions’, *Geophysics* **67**(1), 156–166.
- [17] Higdon, R. L. [1987], ‘Numerical absorbing boundary conditions for the wave equation’, *Mathematics of computation* **49**(179), 65–90.
- [18] Igel, H. [2017], *Computational seismology: a practical introduction*, Oxford University Press.
- [19] Keith Wilson, D. and Thomson, D. W. [1994], ‘Acoustic tomographic monitoring of the atmospheric surface layer’, *Journal of Atmospheric and Oceanic Technology* **11**(3), 751–769.
- [20] Li, J., Gan, L. and Qin, H. [2017], Acoustic velocity tomography for damage detection in concrete, in ‘2017 29th Chinese Control and Decision Conference (CCDC)’, IEEE, pp. 146–149.
- [21] Lines, L. R., Slawinski, R. and Bording, R. P. [1999], ‘A recipe for stability of finite-difference wave-equation computations’, *Geophysics* **64**(3), 967–969.
- [22] Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hükelheim, J., Yount, C., Kelly, P. H., Herrmann, F. J. and Gorman, G. J. [2020], ‘Architecture and performance of devito, a system for automated stencil computation’, *ACM Transactions on Mathematical Software (TOMS)* **46**(1), 1–28.

- [23] *Madagascar* [2021], reproducibility.org.
URL: https://www.reproducibility.org/wiki/Main_Page
- [24] Mariappan, L., Shao, Q., Jiang, C., Yu, K., Ashkenazi, S., Bischof, J. C. and He, B. [2016], ‘Magneto acoustic tomography with short pulsed magnetic field for in-vivo imaging of magnetic iron oxide nanoparticles’, *Nanomedicine: Nanotechnology, Biology and Medicine* **12**(3), 689–699.
- [25] Martin, G. S., Wiley, R. and Marfurt, K. J. [2006], ‘Marmousi2: An elastic upgrade for marmousi’, *The Leading Edge* **25**(2), 156–166.
URL: <https://doi.org/10.1190/1.2172306>
- [26] Meng, J., Atle, A., Calandra, H. and Araya-Polo, M. [2020], ‘Minimod: A finite difference solver for seismic modeling’, *arXiv preprint arXiv:2007.06048* .
URL: <https://arxiv.org/pdf/2007.06048.pdf>
- [27] Michalowicz, B., Raut, E., Kang, Y., Curtis, T., Chapman, B. and Oryspayev, D. [2021], ‘Comparing the behavior of openmp implementations with various applications on two different fujitsu a64fx platforms’, *arXiv preprint arXiv:2106.09787* .
- [28] Micikevicius, P. [2009], ‘3D Finite Difference Computation on GPUs using CUDA’.
- [29] Munk, W. H. and Wunsch, C. I. [1982], ‘Observing the ocean in the 1990s’, *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* **307**(1499), 439–464.
- [30] Pershin, I. S., Levchenko, V. D. and Perepelkina, A. Y. [2019], ‘Performance Limits Study of Stencil Codes on Modern GPGPUs’, *Supercomputing Frontiers and Innovations* **6**(2), 86–101.
- [31] Raut, E., Anderson, J., Araya-Polo, M. and Meng, J. [2021a], Evaluation of distributed tasks in stencil-based application on gpus, *in* ‘2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)’, pp. 45–52.
- [32] Raut, E., Anderson, J., Araya-Polo, M. and Meng, J. [2021b], Porting and evaluation of a distributed task-driven stencil-based application, *in* ‘Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores’, pp. 21–30.
- [33] Raut, E., Meng, J., Araya-Polo, M. and Chapman, B. [2020], Evaluating performance of openmp tasks in a seismic stencil application, *in* ‘International Workshop on OpenMP’, Springer, pp. 67–81.

- [34] Robertsson, J. O. A. [1996], ‘A numerical free-surface condition for elastic/viscoelastic finite-difference modeling in the presence of topography’, *GEOPHYSICS* **61**(6), 1921–1934.
URL: <https://doi.org/10.1190/1.1444107>
- [35] Salari, K. and Knupp, P. [2000], Code verification by the method of manufactured solutions, Technical report, Sandia National Labs., Albuquerque, NM (US); Sandia National Labs
- [36] Schäfer, A. and Fey, D. [2011], ‘High performance stencil code algorithms for gpgpus’, *Procedia Computer Science* **4**, 2027–2036. Proceedings of the International Conference on Computational Science, ICCS 2011.
URL: <https://www.sciencedirect.com/science/article/pii/S1877050911002791>
- [37] Shearer, P. M. [2019], *Introduction to seismology*, Cambridge university press.
- [38] Shiotani, T., Osawa, S., Momoki, S. and Ohtsu, H. [2015], Visualization of damage in rc bridge deck for bullet trains with ae tomography, *in* ‘Advances in Acoustic Emission Technology’, Springer, pp. 357–368.
- [39] Tarantola, A. [1984], ‘Inversion of seismic reflection data in the acoustic approximation’, *Geophysics* **49**(8), 1259–1266.
- [40] Virieux, J. and Operto, S. [2009], ‘An overview of full-waveform inversion in exploration geophysics’, *Geophysics* **74**(6), WCC1—WCC26.
URL: <http://library.seg.org/doi/10.1190/1.3238367>
- [41] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J. et al. [2020], ‘Scipy 1.0: fundamental algorithms for scientific computing in python’, *Nature methods* **17**(3), 261–272.
- [42] *Virtual Earthquake and seismology Research Community e-science environment in Europe (VERCE)*. Project website. [n.d.], <http://www.verce.eu/index.php>. Accessed: 2021-10-06.
- [43] Watanabe, K. [2015], Green’s functions for laplace and wave equations, *in* ‘Integral Transform Techniques for Green’s Function’, Springer, pp. 33–76.
- [44] Xia, H., Liu, G., Huang, X., Guo, L., Yang, Y. and Lu, M. [2017], ‘The forward and inverse problem based on magneto-acoustic tomography with current injection’, *Journal of Biomedical Science and Engineering* **10**(5), 97–105.
- [45] Zhou, K., Meng, X., Sai, R., Grubisic, D. and Mellor-Crummey, J. [2021], ‘An automated tool for analysis and tuning of gpu-accelerated code in hpc applications’, *IEEE Transactions on Parallel and Distributed Systems* **33**(4), 854–865.
