J-WAVE: AN OPEN-SOURCE DIFFERENTIABLE WAVE SIMULATOR

A PREPRINT

Antonio Stanziola*
Dept. of Medical Physics and Biomedical Engineering
University College London
London WC1E 6BT, UK

Simon R. Arridge
Department of Computer Science
University College of London
London WC1E 6BT, UK

Ben T. Cox
Dept. of Medical Physics and Biomedical Engineering
University College London
London WC1E 6BT, UK

Bradley E. Treeby
Dept. of Medical Physics and Biomedical Engineering
University College London
London WC1E 6BT, UK

ABSTRACT

We present an open-source differentiable acoustic simulator, j-Wave, which can solve time-varying and time-harmonic acoustic problems. It supports automatic differentiation, which is a program transformation technique that has many applications, especially in machine learning and scientific computing. j-Wave is composed of modular components that can be easily customized and reused. At the same time, it is compatible with some of the most popular machine learning libraries, such as JAX and TensorFlow. The accuracy of the simulation results for known configurations is evaluated against the widely used k-Wave toolbox and a cohort of acoustic simulation software. j-Wave is available from `https://github.com/ucl-bug/jwave`.

Keywords differentiable simulator · acoustics · machine learning · gpu acceleration · wave equation · Helmholtz equation · jax

1 Motivation and significance

1.1 Background

The accurate simulation of wave phenomena has many interesting applications, from medical physics to seismology and electromagnetics, with the aim of either forecasting, for example, predicting an ultrasound field inside the brain [1], or performing parametric inference, for example, recovering material properties from acoustic measurements using full wave inversion [2]. Many numerical techniques for solving the wave equation have been developed over the years, including pseudospectral algorithms [3], finite differences [4, 5], angular spectrum methods [6] and boundary element methods [7], to name a few.

Recently, there has been a growing body of research at the intersection of numerical simulation and machine learning [8, 9, 10]. The critical observation is that the machine learning community has developed many tools and techniques for high-dimensional inference. In particular, automatic differentiation, the class of algorithms often employed for neural network training and generally for automatic analytical gradient estimation, can be used to differentiate for any continuous parameter involved in a simulator [11, 12]. This enables optimization or parameter identification of all simulator parameters, including the simulated field and other parameters that appear in the governing partial differential equation (PDE), as well as numerical parameters such as the finite difference stencil used to compute gradients.

---

*a.stanziola@ucl.ac.uk

Simulators that allow for automatic differentiation can also be used inside a machine learning model. Examples include implementing implicit layers [13], reinforcement learning [14, 15], parameter identification [16], inverse problems [17], optimal control [18], construction of physics-based loss functions [19, 18, 20], and research into novel discretizations or neural network augmented simulators [21].

1.2 Aim

Here we present j-Wave: a customizable Python simulator, written on top of the JAX library [12] and the discretization framework JaxDF [22], for fast, parallelizable, and differentiable acoustic simulations. j-Wave solves both time-varying and time-harmonic forms of the wave equation with support for multiple discretizations, including finite differences and Fourier spectral methods. Custom discretizations, including those based on neural networks, can also be utilized via the JaxDF framework. The use of the JAX library gives direct support for program transformations, such as automatic differentiation, Single-Program Multiple-Data (SPMD) parallelism, and just-in-time compilation. Lastly, since j-Wave is written in a language that follows the NumPy [23] syntax, it is easy to adapt, enhance or re-implement any simulator stage.

1.3 Related software

There is a range of related software that can be used to simulate acoustic fields, and that can be used as an alternative or to complement j-Wave. In the Julia language, the SciML ecosystem has a variety of tools that can be used to construct differentiable acoustic simulators [9]. In particular, the `ADSeismic.jl` [24] library focuses on seismic wave propagation and several inversion algorithms commonly used in the seismic field, and also includes the support for neural network representation of velocity models [25]. In Python, the Devito package [26] and the recently published Stride [27] library can be used to solve acoustic optimisation problems that scale over large super computing clusters, while SimPEG [28] can be used for geophysical parameter estimation. In JAX, several recent works have developed tools for simulation-based inference and differentiable simulations. These range from integrating JAX with FEniCS for finite elements simulations [29], to differentiable molecular dynamics [30] and fluid dynamics [31].

2 Software description

2.1 Governing equations

j-Wave solves two different forms of the wave equation for time-varying and time-harmonic (i.e., single frequency) problems. For time-varying problems, j-Wave solves a linear system of coupled first-order PDEs that represent the conservation of mass and momentum, and a pressure density relation [32]:

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho_0} \nabla p \tag{1}$$

$$\frac{\partial \rho}{\partial t} = -\rho_0 \nabla \cdot \mathbf{u} + S_M \tag{2}$$

$$p = c_0^2 \rho \ . \tag{3}$$

Here u is the acoustic particle velocity, p is the acoustic pressure, and ρ is the acoustic density. The acoustic medium is characterized by a spatially varying background density ρ_0 and sound speed c_0. The term S_M represents a mass source field.

For time-harmonic simulations, j-Wave solves a form of the Helmholtz equation constructed from the second-order wave equation including Stokes absorption:

$$\frac{1}{c_0^2} \frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{1}{\rho_0} \nabla \rho_0 \cdot \nabla p + \frac{2\alpha_0}{c_0} \frac{\partial^3 p}{\partial t^3} + \frac{\partial S_M}{\partial t} \tag{4}$$

A time-harmonic solution is obtained by substituting $p = Pe^{-i\omega t}$, where ω is frequency in units of rad·s^{-1}, giving

$$-\frac{\omega^2}{c_0^2} P = \nabla^2 P - \frac{1}{\rho_0} \nabla \rho_0 \cdot \nabla P + \frac{2i\omega^3 \alpha_0}{c_0} P - i\omega S_M. \tag{5}$$

This equation accounts for acoustic absorption of the form $\alpha = \alpha_0 \omega^2$, where the absorption coefficient prefactor α_0 has units of Np(rad/s)$^{-2}$m^{-1}.

2.2 Numerical methods

Solvers for the two governing equations given in Sec. 2.1 are constructed using JaxDF [22]. This is a discretization framework that decouples the mathematical definition of the problem from the underlying discretization. Currently, implementations of the differential operators are available for spectral and finite difference discretizations on a regular Cartesian grid. Alternatively, the user can provide a custom discretization compatible with the underlying operations required by the PDEs. That is, only linear discretizations are compatible with time-stepping and Krylov solvers, while non-linear discretizations can be used as physics informed models [10, 9].

For time-varying problems, the wave equation is solved by integrating the first-order system of equations with a semi-implicit first-order Euler integrator. If a spectral or finite difference discretization is used, the fields are defined on a staggered grid to improve long-range accuracy [33] and avoid checker-board artifacts. Radiating boundary conditions are enforced by embedding the effect of a split-field perfectly matched layer (PML) on the time-stepping scheme [3]. When using a Fourier discretisation, j-Wave is equivalent to the implementation in the open-source k-Wave toolbox [33, 32], including the use of a dispersion-corrected finite difference scheme for time integration. The user can further specify a generic measurement operator $f(u, \rho, p)$ to extract instantaneous values from the wavefield at each time step.

For time-harmonic problems, if the underlying discretization of the Helmholtz operator is linear (for example, using Fourier or finite difference methods), the solver is a special case of linear inversion. In this case, j-Wave uses either GMRES or Bi-CGSTAB to compute the solution. These are matrix-free methods, meaning that the numerical matrix that represents the linear operator is never explicitly constructed. Again, radiating boundary conditions are imposed using a PML, by modifying the spatial gradients as in [34]:

$$\frac{\partial}{\partial x} \to \frac{1}{\gamma_x} \frac{\partial}{\partial x} \tag{6}$$

where

$$\gamma_x(x) = \begin{cases} 1, & \text{if } |x| < a \\ 1 + \frac{i}{\omega}\sigma(x) & \text{if } a \le |x| \end{cases}, \tag{7}$$

and σ follows a power-law profile.

2.3 JAX and automatic differentiation

The fundamental idea of j-Wave is to provide a suite of differentiable, parallelizable and customizable acoustic simulators. These requirements are accomplished, in first instance, by writing the simulator in JAX [12], which provides a growing suite of tools for large-scale differentiable computations, including flexible AD, single-device parallelization, multi-device parallelization, and just-in-time compilation [35]. Furthermore, JAX can be considered an adaptable Python compiler that translates and transforms code. This allowed us to define a series of custom classes that can be overwritten or adapted by the user, while still being amendable of transformation.

All forward operators and simulation functions in j-Wave are differentiable through the use of JaxDF using both forward and backward automatic differentiation (AD). This allows the user to obtain gradients for any continuous parameter in the model. This includes both physical parameters, such as the acoustic pressure or sound speed, and numerical parameters, such as the stencils for finite differences or the filters used in Fourier methods. The gradient rules used for computation can also be freely customized.

Solving a linear system, such as the discretized Helmholtz equation, using an iterative solver is also beneficial for gradient calculation. JAX takes advantage of the implicit function theorem to differentiate through fixed-point algorithms with $O(1)$ memory requirements (that is, the intermediate steps of the iterative solver are not stored to compute the gradient). This is a major advantage when gradients of large-scale simulations are needed. See [36] and references therein for a recent discussion of this topic.

2.4 Software architecture

The architecture of j-Wave can be divided into three main kinds of components: objects, operators, and solvers.

Objects: Objects are variables that contain the numerical data that is used during the simulations. They are defined as classes registered to the JAX compiler as a custom pytree node. The primary objects are:

- `Domain`: Defines a regular Cartesian grid with the specified grid spacing and number of points.

Gradients obtained using reverse-mode AD have been shown to be equivalent to the ones obtained using the adjoint-state model [24].
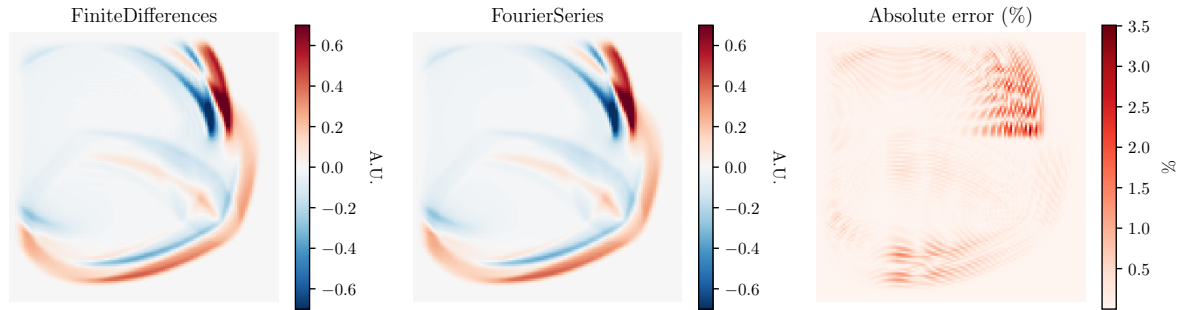
Figure 1: Comparison of the fields produced by j-Wave using 8th order accurate `FiniteDifferences` and `FourierSeries` representations on an initial value problem.

- `Medium`: Defines the `sound_speed` and `density` represented on the specified `domain` along with the `pml_size`.
- `Sources`: Defines the `positions` and `signals` for time varying mass sources within the specified `domain`.
- `Sensors`: Defines the `positions` of detectors placed on the grid.
- `TimeAxis`: Defines the time steps used for time-varying simulations.

Objects can be used as input variables to any JAX function and gradients can be taken with respect to their continuous parameters. They can be unpacked into their constituent numpy-like arrays using the `jax.tree_util.tree_flatten` utility and constructed inside pure functions.

Some parameters are defined as `Field` objects from JaxDF which define underlying discretizations. This includes `medium.sound_speed` and the initial conditions p0 and u0. The discretization used for the input objects governs the discretization used during the calculations. Currently, JaxDF supports `FourierSeries`, `FiniteDifferences` and `Continuous` discretizations. However, it is straightforward to define custom field discretizations which are automatically compiled into their corresponding numerical implementations.

Operators: Operators are defined via JaxDF and implement a numerical algorithm that translates a symbolic operator into its corresponding numerical implementation, for a given type of input discretization. The implementation of the same operator for different discretizations is done using multiple-dispatch via `plum` [37], a programming technique that has been heavily popularized by C#, Lisp and Julia [38], using the `operator` decorator of JaxDF.

For example, a custom Laplacian operator for a 1D `FiniteDifferences` field can be implemented using type hints.

```
@operator                                                              1
def laplacian(u: FiniteDifferences, params=[1, -2, 1]):                2
  k = params                                                           3
  _u = u.on_grid                                                       4
  _u = jnp.pad(_u, (1,1), 'constant', 0)                               5
  v = k[0]*_u[:-2] + k[1]*_u[1:-1] + k[0]*_u[2:]                       6
  return u.replace_params(v), params                                   7
```

Every function that uses the `laplacian` function will then utilize the custom user implementation if the input field is of the type `FiniteDifferences`.

Solvers: There are two main solvers in j-Wave which solve the equations outlined in Sec. 2.1. These are also implemented as operators for convenience.

- `simulate_wave_propagation`: Takes a `medium` object (which internally defines the `Domain`), along with `Sources`, `Sensors`, and `TimeAxis` objects, and initial conditions p0 and u0 if non-zero, and computes the time varying acoustic field over the specified domain.
- `helmholtz_solver`: Takes a `medium` object (which internally defines the `Domain`), `source` field, and frequency `omega` and computes the complex field over the specified domain. The `source` field for the `helmholtz_solver` is a `Field` defined over the entire domain, and can be extracted from `Sources` objects.
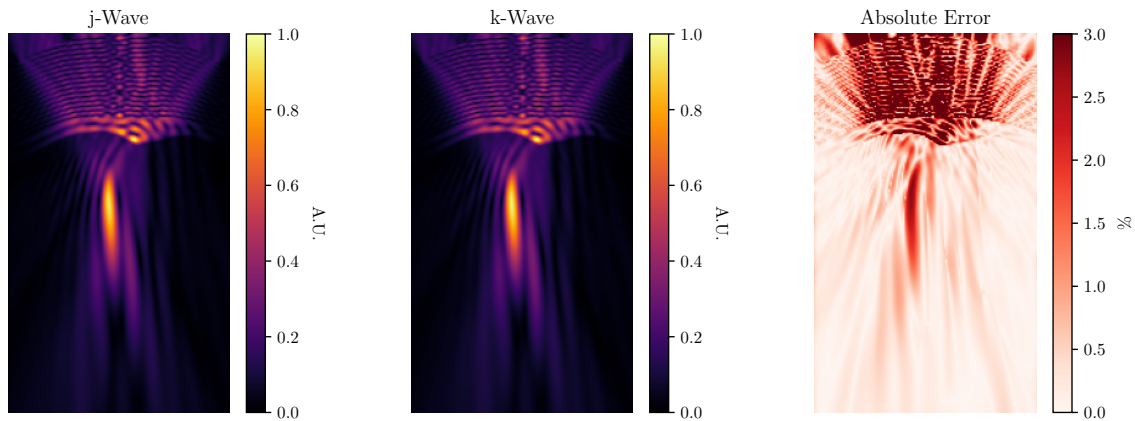
Figure 2: Comparison of the field amplitudes predicted by j-Wave and k-Wave for a focused transducer after propagation through an aberrating skull layer. Adapted from [1].

Simulations using these functions can be performed on CPU, GPUs, and TPUs, with efficient just-in-time compilation, natively compatible with the JAX ecosystem. The functions are also amendable to same-device or multiple-devices parallelization, via the JAX decorators `vmap` and `pmap` [12]. Check-pointing can also be applied at each step to reduce the memory requirements for back-propagation.

2.5 Accuracy

The accuracy of the pseudo-spectral and finite difference solvers has been evaluated both for time-varying and for time-harmonic problems. In the first case, the pseudo-spectral numerical solver is equivalent to k-Wave [33, 32] and numerical simulations agree to machine precision. When finite difference methods are employed, the simulation error is dependent on many factors, other than the implementation itself, such as number of grid points per wavelength, the finite difference coefficients, etc. An illustrative comparison of the wavefields produced for an initial value problem in a medium with a heterogeneous sound speed is shown in Fig. 1.

For the Helmholtz equation, a comprehensive comparison of j-Wave against other wave models (including k-Wave) was conducted as part of the inter-comparison effort described in [1]. For homogeneous material properties, the maximum difference against k-Wave is typically much less than 1%. For heterogeneous properties, the difference depends on which parameters are heterogeneous and the strength of the heterogeneity. Differences are slightly larger for a heterogeneous density (compared to heterogeneous sound speed or absorption). This is likely due to the different way the ambient density term is treated and evaluated on a staggered grid between the two softwares. A representative example showing results for a 3D simulation using j-Wave and k-Wave is given in Fig. 2. This example includes a bone layer with an incident field produced by a focused transducer driven at 500kHz (Benchmark 7 of [1]). In this case, the difference between the two simulations inside the brain is within 3%.

3 Illustrative Examples

3.1 Initial value problems and image reconstruction using time reversal

To demonstrate the process of defining and running a simulation using j-Wave, we start with a simple initial value problem in a homogeneous medium as encountered in, e.g., photoacoustics [39]. Similarly to k-Wave [33], j-Wave requires the user to specify a computational domain where the simulation takes place. This is done using the `Domain` data class inherited from JaxDF as shown in Listing 1. The inputs for the constructor are the size of the domain in grid points in each spatial direction and the corresponding discretization step.

```
from jwave.geometry import Domain                                              1
                                                                               2
N, dx = (128, 128), (0.1e-3, 0.1e-3)                                           3
domain = Domain(N, dx)                                                         4
```

Listing 1: Defining the simulation domain.

The next step is to define the medium properties. This is done using the `Medium` class as shown in Listing 2.

```
from jwave.geometry import Medium                                              1
                                                                               2
medium = Medium(domain=domain, sound_speed=1500.0)                             3
```

Listing 2: Defining the medium properties.

For time-varying problems, a `TimeAxis` object also needs to be defined, which sets the time steps used in the time-stepping scheme of the numerical simulation. This object can be constructed from the medium for a given Courant-Friedrichs-Lewy (CFL) number as shown in Listing 3 to ensure that the time-stepping scheme is stable

```
from jwave.geometry import TimeAxis                                            1
                                                                               2
time_axis = TimeAxis.from_medium(medium, cfl=0.3)                             3
```

Listing 3: Defining the time axis.

The next optional step is to define a `Sensors` object. This is done using the `Sensors` class as shown in Listing 4, which defines the grid points within the domain where the field values are returned (custom sensor definitions can also be used). If no sensors are defined, the code returns a `Field` for each time-step.

```
from jwave.geometry import _points_on_circle, Sensors                          1
                                                                               2
num_sensors, radius, center = 32, 40, (64, 64)                                3
x, y = _points_on_circle(num_sensors, radius, center)                         4
sensors = Sensors(positions=(jnp.array(x), jnp.array(y)))                      5
```

Listing 4: Defining sensors.

Finally, the initial pressure distribution must be defined. This is done by populating a `jax.numpy.ndarray` the same size as the domain, and then passing this to the appropriate discretization. In Listing 5, the initial pressure is set to the weighted sum of four binary disks and defined as a `FourierSeries` field. The field information is used when calling operators to choose the correct numerical implementations. The simulation setup is depicted in Fig. 3 (a).

```
from jwave.geometry import _circ_mask                                          1
from jwave import FourierSeries                                                2
                                                                               3
mask1 = _circ_mask(N, 8, (50,50))                                             4
mask2 = _circ_mask(N, 5, (80,60))                                             5
mask3 = _circ_mask(N, 10, (64,64))                                            6
mask4 = _circ_mask(N, 30, (64,64))                                            7
p0 = 5.*mask1 + 3.*mask2 + 4.*mask3 + 0.5*mask4                               8
p0 = FourierSeries(p0, domain)                                                9
```

Listing 5: Defining the initial pressure distribution as a Fourier series Field.

To run the simulation, the solver `simulate_wave_propagation` is called with the appropriate inputs as shown in Listing 6. Here, a wrapper is defined around it, to highlight how to create arbitrary callables that are just-in-time compiled using `jax.jit`. The recorded acoustic signals are shown in Fig. 3 (b).

```
from jwave.acoustics import simulate_wave_propagation     1
                                                           2
@jit                                                       3
def compiled_simulator(medium, p0):                        4
  return simulate_wave_propagation(                        5
    medium, time_axis, p0=p0, sensors=sensors)             6
                                                           7
sensors_data = compiled_simulator(medium, p0)              8
```

Listing 6: Just-in-time compiling and running the simulation.

3.2 Automatic differentiation

As mentioned, gradients can be evaluated with respect to any input parameters: all that is needed is to define a scalar loss function. In Listing 7, the use of the wave equation adjoint as a simple imaging algorithm for the forward problem defined in Sec. 3.1 is demonstrated following the discretize-then-optimize approach [40, 9]. Note that the user can always define a custom adjoint function for the forward operator if required.

Gradients for the initial pressure alone can be easily computed by wrapping a new function around the simulator and using the `jax.grad` decorator. In this example, noise is added to the data before inverting the model.

```
def solver(p0):                                            1
  return simulate_wave_propagation(                        2
    medium, time_axis, p0=p0, sensors=sensors)             3
                                                           4
@jit  # Compile the whole algorithm                        5
def lazy_imaging_algorithm(measurements):                  6
  # Mask out elements outside the sensors ring             7
  mask = _circ_mask(N, 39, (64, 64))                       8
  mask = np.expand_dims(mask, -1)                          9
                                                          10
  def mse_loss(p0, measurements):                         11
    p0 = p0.replace_params(p0.params * mask)              12
    p_pred = solver(p0)                                   13
    return 0.5 * jnp.sum((p_pred - measurements)**2)      14
                                                          15
  # Start from an empty field                             16
  p0 = FourierSeries.empty(domain)                        17
  # Take the gradient                                     18
  p_grad = grad(mse_loss)(p0, measurements)               19
  return -p_grad                                          20
                                                          21
# Reconstruct initial pressure distribution               22
recon_image = lazy_time_reversal(noisy_data)              23
```

Listing 7: Use of the adjoint model as a simple imaging algorithm.

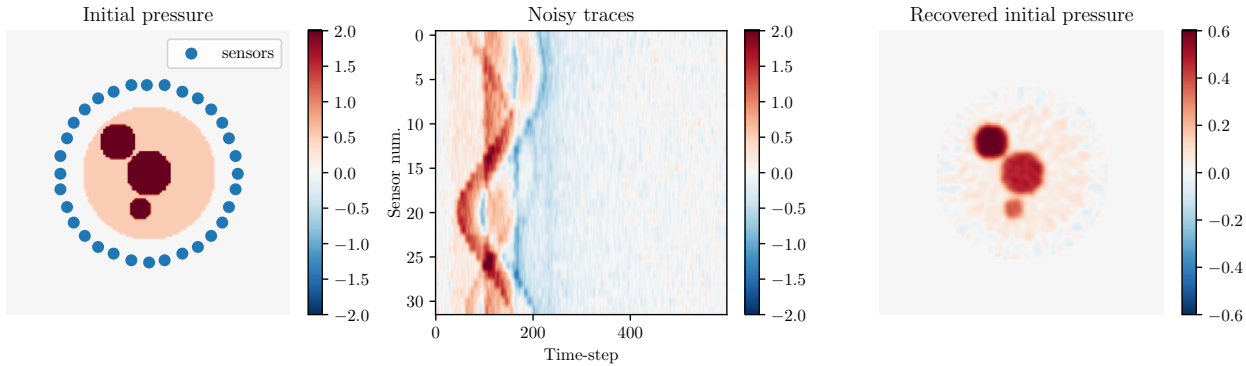The reconstructed initial pressure is shown in Fig. 3 (c).

Figure 3: Example of workflow to simulate an initial value problem and invert it using automatic differentiation. From left to right: Simulation setup; Recorded acoustic signals with additive colored noise; Reconstructed initial pressure distribution from noisy data.

3.3 Prototyping full-wave inversion algorithms

```python
from jwave.signal_processing import analytic_signal

def loss_func(params, source_num):
  # This contains the simulator function
  p = single_source_simulation(get_sound_speed(params), source_num)

  # Get envelopes of data and simulated signals
  p = jnp.abs(analytic_signal(p, 0))
  pred = jnp.abs(analytic_signal(p_data[source_num], 0))

  # MSE on envelopes
  return jnp.sum(jnp.abs(p - pred)**2)

loss_with_grad = jax.value_and_grad(loss_func)
```

Listing 8: Defining an objective function for full-wave inversion.

One of the most exciting features of j-Wave is its (almost) total differentiability. Besides applications in machine learning, differentiability means that full waveform inversion methods can be easily prototyped. For example, to mitigate cycle skipping it has been proposed to use an ℓ_2 loss on the modulus of the complex analytic signal associated with the data residual [41, 42]. This can be implemented by defining an appropriate objective function as shown in Listing 8.

```python
@jax.jit
def update(opt_state, key, k):
  v = get_params(opt_state)
  src_num = random.choice(key, num_sources)

  loss_with_grad = jax.value_and_grad(loss_func, argnums=0)
  lossval, gradient = loss_with_grad(v, src_num)

  gradient = smooth_fun(gradient)
  return lossval, update_fun(k, gradient, opt_state)
```

Listing 9: Gradient descent using AD.

Because it is possible to differentiate through arbitrary computations, evaluating the gradient of this expression is done using backward-mode AD. Low-pass filtering of the FWI speed of sound gradients can also be used to improve the convergence towards the true speed of sound distribution [43]. Again, we can seamlessly include smoothing of the gradients in the update function that is run at each iteration of gradient descent as shown in Listing 9.
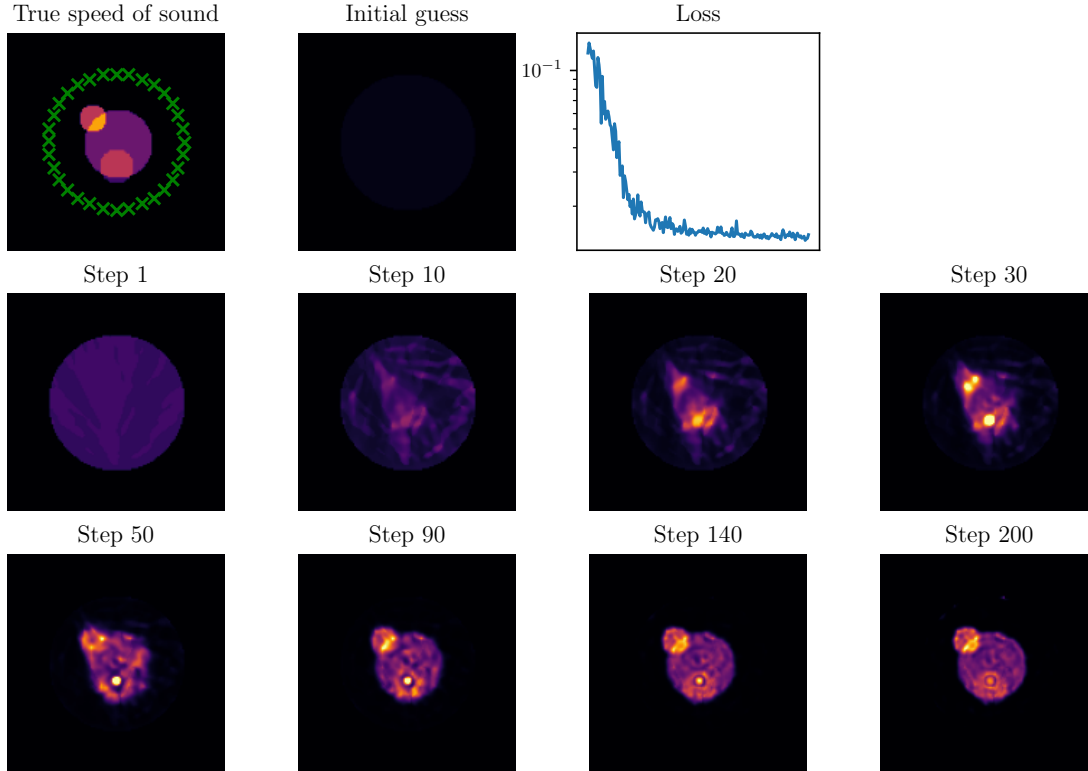
Figure 4: Full wave inversion using an envelope-based objective function and speed of sound gradient smoothing.

The results of this FWI algorithm on a noisy synthetic dataset are given in Fig. 4. Note that this example is only intended to highlight the ability to take gradients of arbitrary computations using a discretize-then-optimize approach.

3.4 Focusing of time-harmonic simulations

As a final example, we demonstrate the differentiability of the time-harmonic solver. We transmit waves from a set of n transducers, that act as monopole sources: that means that we can define a complex weighting vector, that defines the amplitude and phase of the sources

$$\mathbf{a} = (a_0, \ldots, a_n), \qquad a_i \in \mathbb{C}, \; \|a_i\| < 1 \tag{8}$$

such that $\rho(\mathbf{a})$ is the transmitted wavefield. The unit norm constraint is needed to enforce the fact that each transducer has an upper limit on the maximum power it can transmit. One could use several methods to represent this vector and its constraint. Here, we use the following parameterization:

$$a_j(\rho_j, \theta_j) = \frac{e^{i\theta_j}}{1 + \rho_j^2}, \tag{9}$$

where ρ_j and θ_h are real variables.

Often, one wants to find an apodization vector which returns a field having certain properties. For example, in transcranial neurostimulation one may want to maximize the acoustic power delivered to a certain location: this is the setup that we'll use in this example (see Fig. 5 (a)).
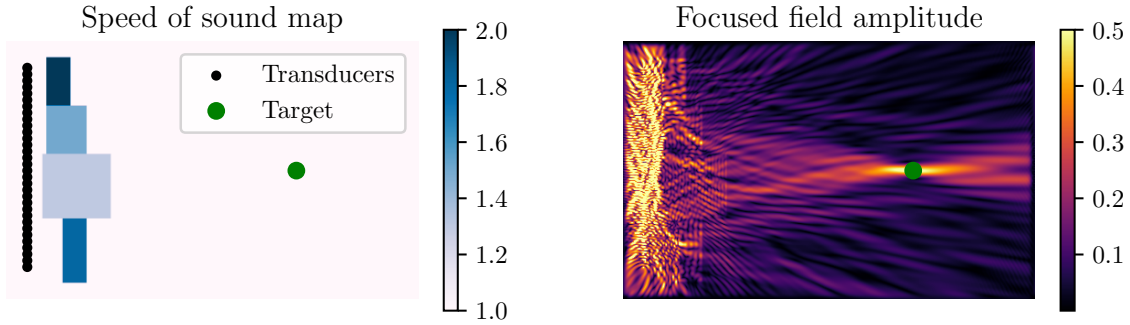
9

Figure 5: Example where the differentiability of the time-harmonic simulator is used. (a) Simulation setup, with a line of point transducers, heterogeneous sound speed and a focusing target; (b) The amplitude of the acoustic field after optimizing the transmit apodization.

Let's call $\mathbf{p} \in \mathbb{R}^2$ the point where we want to maximize the wavefield. For a field $\phi(\mathbf{x}, \mathbf{a})$ generated by the apodization $\mathbf{a}$, the optimal apodization is then given by

$$\hat{\mathbf{a}} = \arg\max_{\mathbf{a}} \|\phi(\mathbf{p}, \mathbf{a})\|. \qquad (10)$$

This defines the loss function that we are going to minimize using gradient descent. The full code for this example is given in the notebook `helmholtz_solver_differentiable.ipynb`, in the examples folder. The resulting wavefield after the optimization is shown in Fig. 5 (b).

4 Impact

j-Wave combines several ideas from the machine learning and inverse problems communities, and can be used to investigate numerical and physical problems revolving around acoustic phenomena. The software is open-source and is based on JAX, which uses an interface that closely follows the widely used NumPy package [23]. This means that interested researchers can customize the software to their needs using a familiar syntax.

As a forward solver, j-Wave can be used as a simple pseudo-spectral acoustic simulator to perform numerical acoustic experiments. The software can simulate wave propagation in homogeneous and heterogeneous media, both in the frequency domain and in the time domain.

The differentiability of the solver can be exploited for a variety of tasks. By taking gradients with respect to the acoustic parameters, j-Wave can perform discrete sensitivity analyses or can be used to learn machine-learning models that perform model-based image inversion. Similarly, gradients with respect to the source parameters can be used for model-based optimal control and training reinforcement learning agents that interact with an acoustic setup.

j-Wave as a differentiable forward model can also be exploited for uncertainty quantification. Besides Monte Carlo methods that can be accelerated in j-Wave using single-device and multiple-device parallel transformations, there is a growing body of techniques that are being developed to exploit simulation gradients for simulation-based inference [8, 44]. For example, in [45], the use of linear uncertainty propagation (LUP) was proposed as a meta-programming method to endow arbitrary (differential) simulations with uncertainty propagation in the Julia language [46]. Supporting forward automatic differentiation allows LUP to be implemented with minimal memory requirements for simulations that depend on a small number of parameters (e.g., uncertainty on the background speed of sound).

Since the operators relevant for acoustic simulations are implemented with JaxDF [22], it is possible to experiment with arbitrary discretizations that contain tunable parameters. This could be leveraged for a variety of tasks such as reduction of memory requirements, computational acceleration, or parameter inference. This is further aided by the possibility of overriding the behaviour of operators for existing or user-defined discretizations. For example, a similar approach has been used recently in computational fluid dynamics, where the authors trained a neural network-based adaptive finite-difference scheme to perform accurate simulations on coarser collocation grids [31]. Alternatively, one could employ learned error-correction schemes [21], directly optimize the stencils of a finite difference scheme [47], or learn a preconditioner for the discretized Helmholtz equation [48].

Operators that represent a PDE, such as the Helmholtz operator, can also be constructed for arbitrary nonlinear discretizations, allowing the application of Physics Informed Neural Networks to solve the acoustic problem [10].

5 Conclusions

An open-source differentiable acoustic simulator called j-Wave is presented that solves both time-harmonic and time-varying forms of the wave equation. The simulator is written in JAX and is compatible with machine learning libraries. Furthermore, it provides a differentiable implementation of the time-harmonic acoustic operator (Helmholtz operator) that can be used either with both linear and non-linear arbitrary discretizations, including ones depending on a set of tunable parameters. We expect j-Wave to be a useful tool for a wide range of acoustic-related lines of research: from the investigation of numerical algorithms and machine learning ideas, to the design of acoustic imaging techniques and materials.

6 Conflict of Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

References

[1] J.-F. Aubry, O. Bates, C. Boehm, K. B. Pauly, D. Christensen, C. Cueto, P. Gelat, L. Guasch, J. Jaros, Y. Jing, et al., Benchmark problems for transcranial ultrasound simulation: Intercomparison of compressional wave models, arXiv preprint arXiv:2202.04552 (2022).

[2] J. Virieux, S. Operto, An overview of full-waveform inversion in exploration geophysics, Geophysics 74 (6) (2009) WCC1–WCC26.

[3] M. Tabei, T. D. Mast, R. C. Waag, A k-space method for coupled first-order acoustic propagation equations, The Journal of the Acoustical Society of America 111 (1) (2002) 53–63.

[4] G. F. Pinton, J. Dahl, S. Rosenzweig, G. E. Trahey, A heterogeneous nonlinear attenuating full-wave model of ultrasound, IEEE transactions on ultrasonics, ferroelectrics, and frequency control 56 (3) (2009) 474–488.

[5] S. Pichardo, C. Moreno-Hernández, R. A. Drainville, V. Sin, L. Curiel, K. Hynynen, A viscoelastic model for the prediction of transcranial ultrasound propagation: Application for the estimation of shear acoustic properties in the human skull, Physics in Medicine & Biology 62 (17) (2017) 6938.

[6] U. Vyas, D. Christensen, Ultrasound beam simulations in inhomogeneous tissue geometries using the hybrid angular spectrum method, IEEE transactions on ultrasonics, ferroelectrics, and frequency control 59 (6) (2012) 1093–1100.

[7] E. van't Wout, P. Gélat, T. Betcke, S. Arridge, A fast boundary element method for the scattering analysis of high-intensity focused ultrasound, The Journal of the Acoustical Society of America 138 (5) (2015) 2726–2737.

[8] K. Cranmer, J. Brehmer, G. Louppe, The frontier of simulation-based inference, Proceedings of the National Academy of Sciences 117 (48) (2020) 30055–30062.

[9] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, A. Edelman, Universal differential equations for scientific machine learning, arXiv preprint arXiv:2001.04385 (2020).

[10] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, Journal of Computational Physics 378 (2019) 686–707.

[11] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, W. Tebbutt, A differentiable programming system to bridge machine learning and scientific computing, arXiv preprint arXiv:1907.07587 (2019).

[12] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs (2018).
URL http://github.com/google/jax

[13] R. T. Chen, Y. Rubanova, J. Bettencourt, D. Duvenaud, Neural ordinary differential equations, arXiv preprint arXiv:1806.07366 (2018).

[14] M. Lutter, J. Silberbauer, J. Watson, J. Peters, Differentiable physics models for real-world offline model-based reinforcement learning, in: 2021 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2021, pp. 4163–4170.

[15] J. K. Murthy, M. Macklin, F. Golemo, V. Voleti, L. Petrini, M. Weiss, B. Considine, J. Parent-Lévesque, K. Xie, K. Erleben, et al., gradsim: Differentiable simulation for system identification and visuomotor control, in: International Conference on Learning Representations, 2020.

[16] E. Heiden, C. E. Denniston, D. Millard, F. Ramos, G. S. Sukhatme, Probabilistic inference of simulation parameters via parallel differentiable simulation, arXiv preprint arXiv:2109.08815 (2021).

[17] J. Liang, M. Lin, V. Koltun, Differentiable cloth simulation for inverse problems (2019).

[18] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, F. Durand, Difftaichi: Differentiable programming for physical simulation, arXiv preprint arXiv:1910.00935 (2019).

[19] A. Karpatne, W. Watkins, J. Read, V. Kumar, Physics-guided neural networks (pgnn): An application in lake temperature modeling, arXiv preprint arXiv:1710.11431 (2017).

[20] P. Holl, V. Koltun, N. Thuerey, Learning to control pdes with differentiable physics, arXiv preprint arXiv:2001.07457 (2020).

[21] A. Siahkoohi, M. Louboutin, F. J. Herrmann, Neural network augmented wave-equation simulation, arXiv preprint arXiv:1910.00925 (2019).

[22] A. Stanziola, S. Arridge, B. T. Cox, B. E. Treeby, A research framework for writing differentiable pde discretizations in jax, Differentiable Programming workshop at Neural Information Processing Systems 2021 (2021).

[23] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., Array programming with numpy, Nature 585 (7825) (2020) 357–362.

[24] W. Zhu, K. Xu, E. Darve, G. C. Beroza, A general approach to seismic inversion with automatic differentiation, Computers & Geosciences (2021) 104751.

[25] W. Zhu, K. Xu, E. Darve, B. Biondi, G. C. Beroza, Integrating deep neural networks with full-waveform inversion: Reparametrization, regularization, and uncertainty quantification, Geophysics 87 (1) (2021) 1–103.

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

[27] C. Cueto, O. Bates, G. Strong, J. Cudeiro, F. Luporini, Ò. Calderón Agudo, G. Gorman, L. Guasch, M.-X. Tang, Stride: A flexible software platform for high-performance ultrasound computed tomography, Computer Methods and Programs in Biomedicine 221 (2022) 106855. doi:https://doi.org/10.1016/j.cmpb.2022.106855.
URL https://www.sciencedirect.com/science/article/pii/S0169260722002371

[28] R. Cockett, S. Kang, L. J. Heagy, A. Pidlisecky, D. W. Oldenburg, Simpeg: An open source framework for simulation and gradient based parameter estimation in geophysical applications, Computers & Geosciences 85 (2015) 142–154.

[29] I. Yashchuk, Bringing pdes to jax with forward and reverse modes automatic differentiation, in: ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations, 2020.

[30] S. S. Schoenholz, E. D. Cubuk, Jax m.d. a framework for differentiable physics, in: Advances in Neural Information Processing Systems, Vol. 33, Curran Associates, Inc., 2020.

[31] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, S. Hoyer, Machine learning–accelerated computational fluid dynamics, Proceedings of the National Academy of Sciences 118 (21) (2021).

[32] B. E. Treeby, J. Jaros, A. P. Rendell, B. Cox, Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using ak-space pseudospectral method, The Journal of the Acoustical Society of America 131 (6) (2012) 4324–4336.

[33] B. E. Treeby, B. T. Cox, k-wave: Matlab toolbox for the simulation and reconstruction of photoacoustic wave fields, Journal of biomedical optics 15 (2) (2010) 021314.

[34] A. Bermúdez, L. Hervella-Nieto, A. Prieto, R. Rodrı, et al., An optimal perfectly matched layer with unbounded absorbing function for time-harmonic acoustic scattering problems, Journal of Computational Physics 223 (2) (2007) 469–488.

[35] D. Häfner, F. Vicentini, mpi4jax: Zero-copy mpi communication of jax arrays, Journal of Open Source Software 6 (65) (2021) 3419.

[36] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, J. Vert, Efficient and modular implicit differentiation, CoRR abs/2105.15183 (2021). arXiv:2105.15183.

[37] Wessel, F. Vicentini, R. Comelli, invenia blog, wesselb/plum: v1.6 (Jun 2022). doi:10.5281/zenodo.6627180.

[38] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM Review 59 (1) (2017) 65–98.

[39] B. T. Cox, P. C. Beard, Fast calculation of pulsed photoacoustic fields in fluids using k-space methods, The Journal of the Acoustical Society of America 117 (6) (2005) 3616–3627.

[40] J. T. Betts, S. L. Campbell, Discretize then optimize, Mathematics for industry: challenges and frontiers (2005) 140–157.

[41] E. Bedrosian, The analytic signal representation of modulated waveforms, Proceedings of the IRE 50 (10) (1962) 2071–2076.

[42] B. Chi, L. Dong, Y. Liu, Full waveform inversion method using envelope objective function without low frequency data, Journal of Applied Geophysics 109 (2014) 36–46.

[43] T. Alkhalifah, Scattering-angle based filtering of the waveform inversion gradients, Geophysical Journal International 200 (1) (2014) 363–373.

[44] A. R. Gerlach, A. Leonard, J. Rogers, C. Rackauckas, The koopman expectation: An operator theoretic method for efficient analysis and optimization of uncertain hybrid dynamical systems, arXiv preprint arXiv:2008.08737 (2020).

[45] M. Giordano, Uncertainty propagation with functionally correlated quantities, arXiv preprint arXiv:1610.08716 (2016).

[46] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM review 59 (1) (2017) 65–98.

[47] C.-H. Jo, C. Shin, J. H. Suh, An optimal 9-point, finite-difference, frequency-space, 2-d scalar wave extrapolator, Geophysics 61 (2) (1996) 529–537.

[48] Y. Azulay, E. Treister, Multigrid-augmented deep learning preconditioners for the helmholtz equation, in: The Symbiosis of Deep Learning and Differential Equations, 2021.