SoftwareX 22 (2023) 101338

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

# j-Wave: An open-source differentiable wave simulator

Antonio Stanziola [a,*], Simon R. Arridge [b], Ben T. Cox [a], Bradley E. Treeby [a]

[a] Department of Medical Physics and Biomedical Engineering, University College of London, Gower Street, London WC1E 6BT, UK
[b] Department of Computer Science, University College of London, Gower Street, London WC1E 6BT, UK

ARTICLE INFO

*Article history:*
Received 16 August 2022
Received in revised form 22 January 2023
Accepted 7 February 2023

*Keywords:*
Differentiable simulator
Acoustics
Machine learning
GPU acceleration
Wave equation
Helmholtz equation
JAX

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.



## Code metadata

| | |
|---|---|
| Current code version | 0.0.4 |
| Permanent link to code/repository used for this code version | https://github.com/ElsevierSoftwareX/SOFTX-D-22-00241 |
| Legal Code License | LGPL-3.0 |
| Code versioning system used | git |
| Software code languages, tools, and services used | Python 3 |
| Compilation requirements, operating environments & dependencies | jax, plum-dispatch, jaxdf |
| If available Link to developer documentation/manual | https://ucl-bug.github.io/jwave/ |
| Support email for questions | a.stanziola@ucl.ac.uk |

## 1. Motivation and significance

### 1.1. Background

The accurate simulation of wave phenomena has many interesting applications, from medical physics to seismology and electromagnetics. The aim is usually either forecasting, such as predicting an ultrasound field inside the brain [1], or performing parametric inference, such as recovering material properties from acoustic measurements using full-waveform inversion (FWI) [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 [8–10]. 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.

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 [18–20], and research into novel discretizations or neural network augmented simulators [21].

* Corresponding author.
*E-mail address:* a.stanziola@ucl.ac.uk (Antonio Stanziola).

https://doi.org/10.1016/j.softx.2023.101338

### 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, in 1D, 2D and 3D. 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 optimization 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 it with FEniCS for finite elements simulations [29], to differentiable molecular dynamics [30] and fluid dynamics [31] simulators.

## 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 $\rho$ is the acoustic density. The acoustic medium is characterized by a spatially varying background density $\rho_0$ 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 $\omega$ 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 $\alpha_0$ has units of Np(rad/s)$^{-2}$m$^{-1}$.

### 2.2. Numerical methods

Solvers for the two governing equations given in Section 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 [9,10].

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 discretization, j-Wave is equivalent to the implementation in the open-source k-Wave toolbox [32,33], 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 $\sigma$ 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 a flexible automatic differentiation (AD) engine, 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 amenable to transformation.

All forward operators and simulation functions in j-Wave are differentiable through the use of JaxDF using both forward and backward 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.[1]

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.
- `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.

Note, that whenever possible j-Wave uses duck-typing, meaning that each kind of object used only needs to provide specific methods and characteristics to be used. So, for example, a valid source only needs to provide a method `on_grid(n)` that returns the field on the grid at the $n$th iteration, as for example does the `Sources` class. A valid sensor is instead any object that can be called with the signature `(p: Field, u: Field, rho: Field)`, as it is implemented in the `Sensors` class. An example of using the flexibility of duck-typing is in the `Transducer` class, which can work both as a source and as a sensor object.

[1] Gradients obtained using reverse-mode AD have been shown to be equivalent to the ones obtained using the adjoint-state model [24].

**Operators:** Operators are defined via JaxDF and implement a numerical algorithm that translates a mathematical 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, as shown in Listing 1.

```
@operator
def laplacian(u: FiniteDifferences, params
    =[1, -2, 1]):
  # Extract the stencil from the parameters
  k = params
  # Get the field on the domain grid
  _u = u.on_grid
  # Add zero-padding
  _u = jnp.pad(_u, (1,1), 'constant', 0)
  # Apply the stencil
  v = k[0]*_u[:-2] + k[1]*_u[1:-1] + k[0]*_u
     [2:]
  # Update the field values and return with
     the operator parameters
  return u.replace_params(v), params
```

**Listing 1:** Redefining the Laplacian operator for finite differences.

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 Section 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.

Simulations using these functions can be performed on CPUs, GPUs, and TPUs, with efficient just-in-time compilation, natively compatible with the JAX ecosystem. The functions are also amenable to same-device or multiple-device 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 [32,33] and numerical

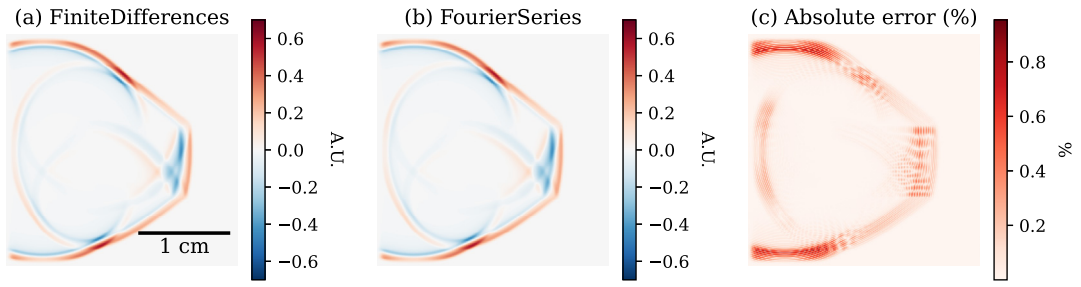


**Fig. 1.** Comparison of the fields produced by j-Wave using 8th order accurate `FiniteDifferences` and `FourierSeries` representations on an initial value problem. The initial pressure distribution is a smoothed disk, the speed of sound of the medium is 1500 m/s with a rectangular heterogeneity of 2300 m/s, and the simulation is run until $t = 8$ μs.

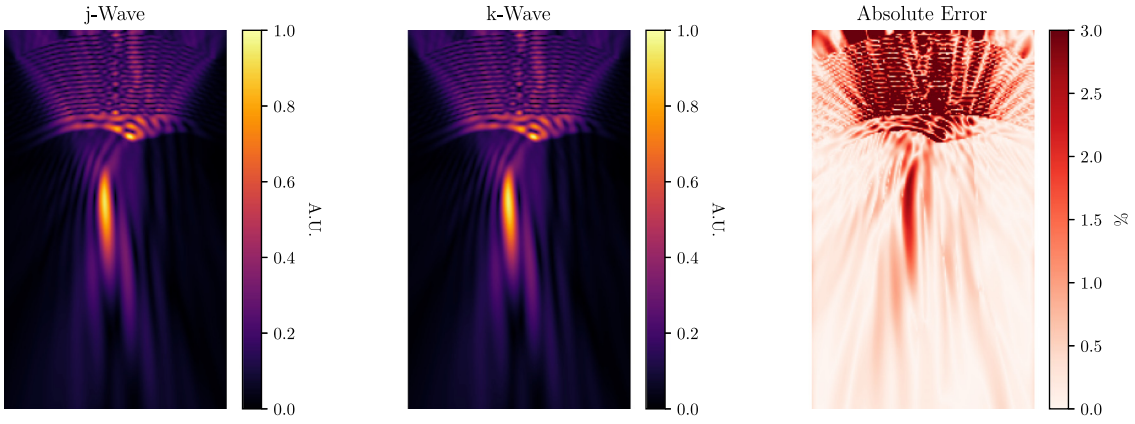


**Fig. 2.** Comparison of the field amplitudes predicted by j-Wave and k-Wave for a focused transducer after propagation through an aberrating skull layer in 3D. *Source:* Adapted from [1].

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 the 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 500 kHz (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 2. 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

N, dx = (256, 256), (0.1e-3, 0.1e-3)
domain = Domain(N, dx)
```

**Listing 2:** Defining the simulation domain.

The next step is to define the medium properties. This is done using the `Medium` class as shown in Listing 3.

```
from jwave.geometry import Medium

medium = Medium(domain=domain, sound_speed
    =1500.0)
```

**Listing 3:** 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 4 to ensure that the time-stepping scheme is stable.

```
from jwave.geometry import TimeAxis

time_axis = TimeAxis.from_medium(medium, cfl=0.3)
```

**Listing 4:** Defining the time axis.

The next optional step is to define some sensors. Any kind of sensor can be used, where a custom sensor is any object or function that can be called. When called, it must take the current velocity, density and pressure fields as inputs, and return a PyTree. For convenience, j-Wave provides the `Sensors` class for defining sensors on grid-points, as shown in Listing 5. If no sensors are defined, the code returns a `Field` for each time-step.

```
from jwave.geometry import _points_on_circle,
    Sensors

num_sensors, radius, center = 48, 100, (128, 128)
x, y = _points_on_circle(num_sensors, radius,
    center)
sensors = Sensors(positions=(jnp.array(x), jnp.
    array(y)))
```

**Listing 5:** 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 6, the initial pressure is set to the weighted sum of four binary disks and defined as a `FourierSeries` field. The disks are generated using the custom function `_circ_mask` which generates circular binary masks of a given radius and location, but any numpy array (however constructed) can be used as the initial pressure. The field information is used when calling operators to choose the correct numerical implementations. The simulation setup is depicted in Fig. 3 (left).

```
from jwave.geometry import _circ_mask
from jwave import FourierSeries

mask1 = _circ_mask(N, 16, (100, 100))
mask2 = _circ_mask(N, 10, (160, 120))
mask3 = _circ_mask(N, 20, (128, 128))
mask4 = _circ_mask(N, 60, (128, 128))
p0 = 5.*mask1 + 3.*mask2 + 4.*mask3 + 0.5*mask4
p0 = FourierSeries(p0, domain)
```

**Listing 6:** 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 7. Here, a wrapper is defined around it, to highlight how to create arbitrary callables that are just-in-time compiled using `jax.jit`. The jit compilation of the entire computational graph makes the solver computationally very efficient: on a modern GPU, the user can expect to run 1000 time-steps on a $128 \times 128$ domain in about 50 ms. The recorded acoustic signals are shown in Fig. 3 (center).

```
from jwave.acoustics import
    simulate_wave_propagation

@jit
def compiled_simulator(medium, p0):
  return simulate_wave_propagation(
    medium, time_axis, p0=p0, sensors=sensors)

sensors_data = compiled_simulator(medium, p0)
```

**Listing 7:** 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 8, the use of the wave equation adjoint as a simple imaging algorithm for the forward problem defined in Section 3.1 is demonstrated following the discretize-then-optimize approach [9,40]. 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):
  return simulate_wave_propagation(
    medium, time_axis, p0=p0, sensors=sensors)

@jit  # Compile the whole algorithm
def lazy_imaging_algorithm(measurements):
  def mse_loss(p0, measurements):
    p_pred = solver(p0)
    return 0.5 * jnp.sum((p_pred - measurements)
    **2)

  # Start from an empty field
  p0 = FourierSeries.empty(domain)
  # Take the gradient
  p_grad = grad(mse_loss)(p0, measurements)
  return -p_grad

# Reconstruct initial pressure distribution
recon_image = lazy_time_reversal(noisy_data)
```

**Listing 8:** Use of the adjoint model as a simple imaging algorithm.

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

### 3.3. Prototyping full-waveform inversion algorithms

One of the most exciting features of j-Wave is the possibility to obtain gradients with respect to any continuous real parameter of the computational graph directly exposed to the user. 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 $\ell_2$ 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 9.

```
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 9:** Defining an objective function for full-wave inversion.

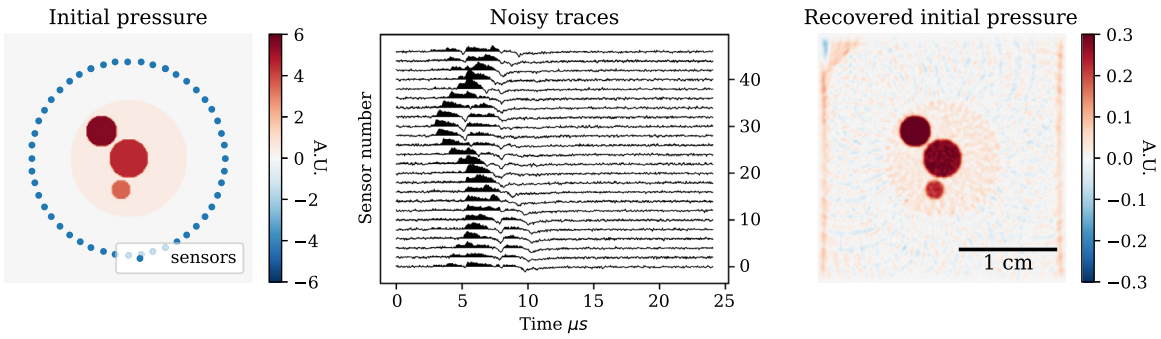


**Fig. 3.** Example of workflow to simulate an initial value problem and invert it using automatic differentiation. From left to right: Simulation setup; Recorded acoustic signals with additive colored noise; Reconstructed initial pressure distribution from noisy data.

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 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 10.

```
@jax.jit
def update(opt_state, key, k):
  # Get the parameters from the optimizer
  v = get_params(opt_state)
  # Sample a random source
  src_num = random.choice(key, num_sources)

  # Find the value of the loss and its gradient
  loss_with_grad = jax.value_and_grad(loss_func,
    argnums=0)
  lossval, gradient = loss_with_grad(v, src_num)

  # Smooth the gradient
  gradient = smooth_fun(gradient)

  # Update the parameters using the gradient and
    return with loss value
  return lossval, update_fun(k, gradient,
    opt_state)
```

**Listing 10:** Gradient descent using AD.

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 another 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 $\phi(\mathbf{a})$ 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 $\rho_j$ and $\theta_h$ 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 will use in this example (see Fig. 5(a)).

Let us 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

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

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).

Time harmonic solvers also benefit from the jit compilation capabilities. For example, the time required to solve a $128 \times 256$ problem on a modern GPU is about 200 ms. However, due to the use of matrix-free operations, currently j-Wave does not implement preconditioning for the solution of the Helmholtz solver. Therefore the performance of the GMRES solver will degrade for large 3D problems. This could be amended by using domain decomposition methods [44], which will be the focus of future package releases.

## 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 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

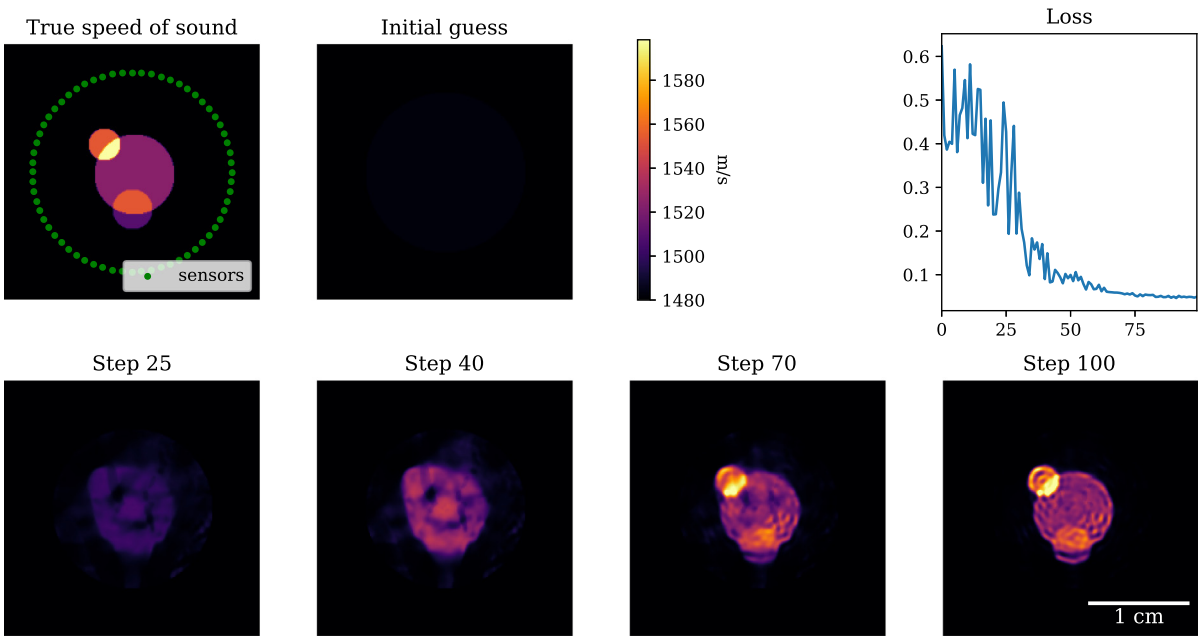


**Fig. 4.** Full -waveform inversion using an envelope-based objective function and speed of sound gradient smoothing. The first row from the left shows the data acquisition setup, the initial guess and the evolution of the loss function for each gradient descent step. The second row shows the estimated speed of sound map at several steps.

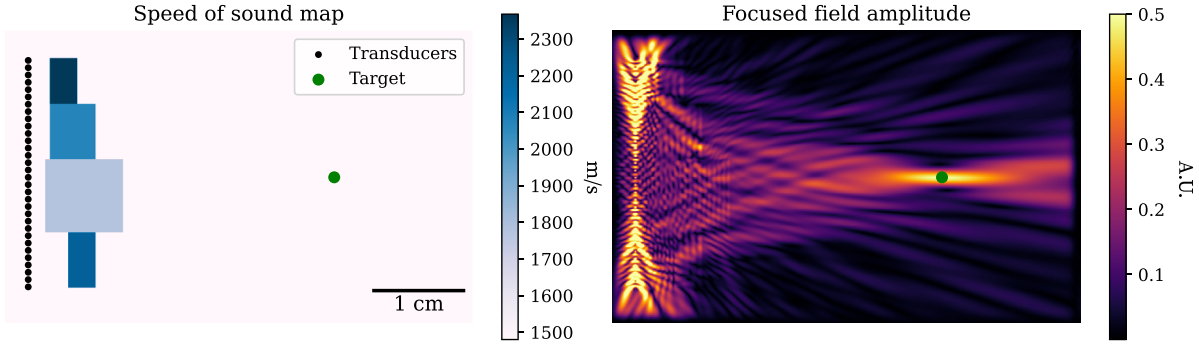


**Fig. 5.** Example where the differentiability of the time-harmonic simulator is used. (a) Simulation setup, with a line of point transducers, heterogeneous sound speed and a focusing target; the sources frequency is 1.7MHz. (b) The amplitude of the acoustic field after optimizing the transmit apodization.

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 [8,45]. For example, in [46], 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 [47]. 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). An example of using LUP with j-Wave is included in one of the example notebooks provided with the package.

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 behavior 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 [48], or learn a preconditioner for the discretized Helmholtz equation [49].

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].

While the software offers a complete set of tools for accurately simulating wave phenomena, the user should remember that it has been designed mainly to allow flexible experimentation in the context of learned algorithms, such as machine learning. Therefore, it is likely prohibitive to use the software out of the box for large-scale problems such as 3D seismic imaging and medical tomography on real data, as it would require a prohibitive amount of memory for a consumer computer. However, it is possible to use the components of j-Wave to research algorithms for reducing memory requirements.

Similarly, in its current state, j-Wave does not support MPI [35], although some operations could be easily implemented using the MPI protocol. Also, domain decomposition methods are

not implemented; hence multi-GPU hosts can only be exploited for the concurrent solution of distinct problems.

Lastly, even though the iterative solvers used for the solution of harmonic problems allow for the use of preconditioners, most of the acoustic operators are implemented as matrix-free and therefore do not allow for immediate applications of preconditioners, especially for the back-propagation step. This means that for large-scale problems, the solvers will require a large number of iterations to converge, which may justify switching to a time-domain solver run to a steady state. We, however, note that the satisfactory preconditioning of the Helmholtz equation is still an active area of research [50], and each computation step in j-Wave can always be overwritten with a more effective algorithm by the user.

## 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, with potential applications in medical ultrasound, non-destructive testing, acoustic material design, seismic modeling and general machine learning research on acoustics. 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 with both linear and nonlinear 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.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council, United Kingdom (EPSRC), UK, grant numbers EP/S026371/1, EP/V026259/1 and EP/T022280/1.

## References

[1] Aubry J-F, Bates O, Boehm C, Pauly KB, Christensen D, Cueto C, et al. Benchmark problems for transcranial ultrasound simulation: Intercomparison of compressional wave models. J Acoust Soc Am 2022;152:1003–19.
[2] Virieux J, Operto S. An overview of full-waveform inversion in exploration geophysics. Geophysics 2009;74(6):WCC1–26.
[3] Tabei M, Mast TD, Waag RC. A k-space method for coupled first-order acoustic propagation equations. J Acoust Soc Am 2002;111(1):53–63.
[4] Pinton GF, Dahl J, Rosenzweig S, Trahey GE. A heterogeneous nonlinear attenuating full-wave model of ultrasound. IEEE Trans Ultrason Ferroelectr Freq Control 2009;56(3):474–88.
[5] Pichardo S, Moreno-Hernández C, Drainville RA, Sin V, Curiel L, Hynynen K. A viscoelastic model for the prediction of transcranial ultrasound propagation: Application for the estimation of shear acoustic properties in the human skull. Phys Med Biol 2017;62(17):6938.
[6] Vyas U, Christensen D. Ultrasound beam simulations in inhomogeneous tissue geometries using the hybrid angular spectrum method. IEEE Trans Ultrason Ferroelectr Freq Control 2012;59(6):1093–100.
[7] van't Wout E, Gélat P, Betcke T, Arridge S. A fast boundary element method for the scattering analysis of high-intensity focused ultrasound. J Acoust Soc Am 2015;138(5):2726–37.
[8] Cranmer K, Brehmer J, Louppe G. The frontier of simulation-based inference. Proc Natl Acad Sci 2020;117(48):30055–62.
[9] Rackauckas C, Ma Y, Martensen J, Warner C, Zubov K, Supekar R, et al. Universal differential equations for scientific machine learning. 2020, arXiv preprint arXiv:2001.04385.
[10] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J Comput Phys 2019;378:686–707.
[11] Innes M, Edelman A, Fischer K, Rackauckas C, Saba E, Shah VB, et al. A differentiable programming system to bridge machine learning and scientific computing. 2019, arXiv preprint arXiv:1907.07587.
[12] Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, et al. [JAX]: composable transformations of Python+NumPy programs. 2018, http://github.com/google/jax.
[13] Chen RT, Rubanova Y, Bettencourt J, Duvenaud D. Neural ordinary differential equations. 2018, arXiv preprint arXiv:1806.07366.
[14] Lutter M, Silberbauer J, Watson J, Peters J. Differentiable physics models for real-world offline model-based reinforcement learning. In: 2021 IEEE international conference on robotics and automation. ICRA, IEEE; 2021, p. 4163–70.
[15] Murthy JK, Macklin M, Golemo F, Voleti V, Petrini L, Weiss M, et al. gradsim: Differentiable simulation for system identification and visuomotor control. In: International conference on learning representations, 2020.
[16] Heiden E, Denniston CE, Millard D, Ramos F, Sukhatme GS. Probabilistic inference of simulation parameters via parallel differentiable simulation. 2021, arXiv preprint arXiv:2109.08815.
[17] Liang J, Lin M, Koltun V. Differentiable cloth simulation for inverse problems. Adv Neural Inf Process Syst 2019;32.
[18] Hu Y, Anderson L, Li T-M, Sun Q, Carr N, Ragan-Kelley J, et al. Difftaichi: Differentiable programming for physical simulation. 2019, arXiv preprint arXiv:1910.00935.
[19] Karpatne A, Watkins W, Read J, Kumar V. Physics-guided neural networks (pgnn): An application in lake temperature modeling. 2017, arXiv preprint arXiv:1710.11431.
[20] Holl P, Koltun V, Thuerey N. Learning to control pdes with differentiable physics. 2020, arXiv preprint arXiv:2001.07457.
[21] Siahkoohi A, Louboutin M, Herrmann FJ. Neural network augmented wave-equation simulation. 2019, arXiv preprint arXiv:1910.00925.
[22] Stanziola A, Arridge S, Cox BT, Treeby BE. A research framework for writing differentiable pde discretizations in jax. In: Differentiable programming workshop at neural information processing systems 2021, 2021.
[23] Harris CR, Millman KJ, Van Der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with numpy. Nature 2020;585(7825):357–62.
[24] Zhu W, Xu K, Darve E, Beroza GC. A general approach to seismic inversion with automatic differentiation. Comput Geosci 2021;104751.
[25] Zhu W, Xu K, Darve E, Biondi B, Beroza GC. Integrating deep neural networks with full-waveform inversion: Reparametrization, regularization, and uncertainty quantification. Geophysics 2021;87(1):1–103.
[26] Lange M, Kukreja N, Louboutin M, Luporini F, Vieira F, Pandolfo V, et al. 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, p. 67–75.
[27] Cueto C, Bates O, Strong G, Cudeiro J, Luporini F, Calderón Agudo Ò, et al. Stride: A flexible software platform for high-performance ultrasound computed tomography. Comput Methods Programs Biomed 2022;221:106855.
[28] Cockett R, Kang S, Heagy LJ, Pidlisecky A, Oldenburg DW. Simpeg: An open source framework for simulation and gradient based parameter estimation in geophysical applications. Comput Geosci 2015;85:142–54.
[29] Yashchuk I. 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] Schoenholz SS, Cubuk ED. JAX m.d. a framework for differentiable physics. In: Advances in neural information processing systems. Vol. 33, Curran Associates, Inc.; 2020.
[31] Kochkov D, Smith JA, Alieva A, Wang Q, Brenner MP, Hoyer S. Machine learning–accelerated computational fluid dynamics. Proc Natl Acad Sci 2021;118(21).
[32] Treeby BE, Jaros J, Rendell AP, Cox B. Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using ak-space pseudospectral method. J Acoust Soc Am 2012;131(6):4324–36.
[33] Treeby BE, Cox BT. K-wave: Matlab toolbox for the simulation and reconstruction of photoacoustic wave fields. J Biomed Opt 2010;15(2):021314.
[34] Bermúdez A, Hervella-Nieto L, Prieto A, Rodrı R, et al. An optimal perfectly matched layer with unbounded absorbing function for time-harmonic acoustic scattering problems. J Comput Phys 2007;223(2):469–88.

[35] Häfner D, Vicentini F. Mpi4jax: Zero-copy mpi communication of jax arrays. J Open Source Softw 2021;6(65):3419. http://dx.doi.org/10.21105/joss.03419.
[36] Blondel M, Berthet Q, Cuturi M, Frostig R, Hoyer S, Llinares-López F, et al. Efficient and modular implicit differentiation. 2021, arXiv preprint arXiv:2105.15183.
[37] Wessel, Vicentini F, Comelli R. invenia blog, wesselb/plum: v1.6. 2022, http://dx.doi.org/10.5281/zenodo.6627180.
[38] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM Rev 2017;59(1):65–98.
[39] Cox BT, Beard PC. Fast calculation of pulsed photoacoustic fields in fluids using k-space methods. J Acoust Soc Am 2005;117(6):3616–27.
[40] Betts JT, Campbell SL. Discretize then optimize. In: Mathematics for industry: challenges and frontiers. 2005, p. 140–57.
[41] Bedrosian E. The analytic signal representation of modulated waveforms. Proc IRE 1962;50(10):2071–6.
[42] Chi B, Dong L, Liu Y. Full waveform inversion method using envelope objective function without low frequency data. J Appl Geophys 2014;109:36–46.
[43] Alkhalifah T. Scattering-angle based filtering of the waveform inversion gradients. Geophys J Int 2014;200(1):363–73.
[44] Royer A, Geuzaine C, Béchet E, Modave A. A non-overlapping domain decomposition method with perfectly matched layer transmission conditions for the helmholtz equation. Comput Methods Appl Mech Engrg 2022;395:115006.
[45] Gerlach AR, Leonard A, Rogers J, Rackauckas C. The Koopman expectation: An operator theoretic method for efficient analysis and optimization of uncertain hybrid dynamical systems. 2020, arXiv preprint arXiv:2008.08737.
[46] Giordano M. Uncertainty propagation with functionally correlated quantities. 2016, arXiv preprint arXiv:1610.08716.
[47] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM Rev 2017;59(1):65–98.
[48] Jo C-H, Shin C, Suh JH. An optimal 9-point, finite-difference, frequency-space, 2-d scalar wave extrapolator. Geophysics 1996;61(2):529–37.
[49] Azulay Y, Treister E. Multigrid-augmented deep learning preconditioners for the helmholtz equation. In: The symbiosis of deep learning and differential equations, 2021.
[50] Gander MJ, Zhang H. A class of iterative solvers for the helmholtz equation: Factorizations, sweeping preconditioners, source transfer, single layer potentials, polarized traces, and optimized schwarz methods. Siam Rev 2019;61(1):3–76.