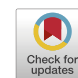
Computer Methods and Programs in Biomedicine 221 (2022) 106855

Contents lists available at ScienceDirect

Computer Methods and Programs in Biomedicine

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

Stride: A flexible software platform for high-performance ultrasound computed tomography



Carlos Cueto [a,*], Oscar Bates [a], George Strong [b], Javier Cudeiro [b], Fabio Luporini [c], Òscar Calderón Agudo [b], Gerard Gorman [b], Lluis Guasch [b,**], Meng-Xing Tang [a,**]

[a] Department of Bioengineering, Imperial College London,London,SW7 2AZ,United Kingdom
[b] Department of Earth Science and Engineering, Imperial College London, London, SW7 2AZ, United Kingdom
[c] Devito Codes, London, United Kingdom

ARTICLE INFO

Article history:
Received 9 November 2021
Revised 25 April 2022
Accepted 3 May 2022

ABSTRACT

Background and objective: Advanced ultrasound computed tomography techniques like full-waveform inversion are mathematically complex and orders of magnitude more computationally expensive than conventional ultrasound imaging methods. This computational and algorithmic complexity, and a lack of open-source libraries in this field, represent a barrier preventing the generalised adoption of these techniques, slowing the pace of research, and hindering reproducibility. Consequently, we have developed Stride, an open-source Python library for the solution of large-scale ultrasound tomography problems.

Methods: On one hand, Stride provides high-level interfaces and tools for expressing the types of optimisation problems encountered in medical ultrasound tomography. On the other, these high-level abstractions seamlessly integrate with high-performance wave-equation solvers and with scalable parallelisation routines. The wave-equation solvers are generated automatically using Devito, a domain-specific language, and the parallelisation routines are provided through the custom actor-based library Mosaic.

Results: We demonstrate the modelling accuracy achieved by our wave-equation solvers through a comparison (1) with analytical solutions for a homogeneous medium, and (2) with state-of-the-art modelling software applied to a high-contrast, complex skull section. Additionally, we show through a series of examples how Stride can handle realistic numerical and experimental tomographic problems, in 2D and 3D, and how it can scale robustly from a local multi-processing environment to a multi-node high-performance cluster.

Conclusions: Stride enables researchers to rapidly and intuitively develop new imaging algorithms and to explore novel physics without sacrificing performance and scalability. This will lead to faster scientific progress in this field and will significantly ease clinical translation.

© 2022 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

1. Introduction

Ultrasound computed tomography techniques such as full-waveform inversion (FWI) have the potential to produce high-resolution, 3D reconstructions of tissues such as the breast [1,2], the limbs [3], or the adult human brain [4]. However, generalised adoption of these techniques is hindered by the fact that tomography algorithms are computationally demanding and algorithmically complex, while existing medical tomography codes are, as far as we are aware, closed source, difficult to maintain, and slow to adapt to new research.

FWI is a technique, originally developed in the field of geophysics, that produces reconstructions of tissue properties by solving an associated inverse problem. FWI is computationally expensive because, for realistic 3D problems, it requires the solution of thousands of partial-differential equations (PDEs) and the storage of hundreds of gigabytes of memory at every iteration in order to estimate billions of parameters. At the same time, FWI is algorithmically challenging due to the non-linear, non-convex nature of the inverse problem being solved. Therefore, any software for solving FWI problems has to address its computational and algorithmic needs, but should also emphasise the high-level, problem-specific

* Corresponding author.
** Principal corresponding author.
E-mail addresses: c.cueto@imperial.ac.uk (C. Cueto), l.guasch08@imperial.ac.uk (L. Guasch), mengxing.tang@imperial.ac.uk (M.-X. Tang).

https://doi.org/10.1016/j.cmpb.2022.106855
0169-2607/© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

abstractions that are necessary to ease the adoption of these tomographic techniques.

In the fields of geophysics and seismic exploration, different approaches have been taken by open-source libraries to solve these issues. On one hand, libraries like Madagascar [5], SimPEG [6], and PySIT [7], have managed to provide flexibility and high-level abstractions, but have done so at the expense of performance. On the other hand, libraries like SAVA [8] and JavaSeis [9] have focused on performance at the expense of flexibility and extensibility. As a way to bridge the gap between these two extremes, libraries such as SeisFlows and Pyatoa [10,11], jInv [12], and Waveform [13] provide flexible interfaces in high-abstraction languages like Python, Julia or MATLAB that interface with high-performance, hand-tuned solvers. LASIF and Inversionson [14,15] have followed a similar approach by providing modular seismic work-flow management that wraps a high-performance tomography solver.

Recently JUDI [16], written in Julia, has gone a step further by providing high-level abstractions in a modern language together with high-performance solvers that are automatically generated by the domain-specific language (DSL) Devito [17,18]. Automatic code generation for solvers is increasingly important with an ever growing number of specialised architectures, from traditional central processing units (CPUs) to graphical processing units (GPUs) and field-programmable gate arrays (FPGAs), as well as associated parallel programming languages (Cuda, OpenACC, etc.). Fine tuning codes for each of them by hand would be a daunting task for most researchers, whereas DSLs like Devito can generate code that is automatically tuned for each target architecture and parallel language. In doing so, DSLs also increase productivity by simplifying the implementation of new types of physics and discretisations.

The high computational complexity of FWI also requires, for realistic problems, that codes can be deployed to specialised high-performance computing (HPC) systems like multi-node clusters or cloud computing services. This represents a further barrier for domain scientists, who are generally not proficient in the use of HPC systems. Of the reviewed geophysical and seismic libraries, only some of them, such as LASIF and Inversionson, and SeisFlows and Pyatoa, have been designed with HPC deployment and scaling in mind.

Here, we present Stride, an open-source Python library for medical ultrasound tomography that emphasises flexibility and modularity, high performance, and scalability. It achieves this, firstly, through high-level, domain-specific abstractions and heuristics. Secondly, by integrating with the automatic code generation library Devito. Finally, we introduce a parallelisation library for seamless HPC deployment and scaling. Stride is available on GitHub[1] and a complete documentation of its interfaces is available online.[2]

The remaining of this paper is structured as follows: in Section 2, we will present an overview of the structure of Stride, followed by a more detailed exploration of each of its components with accompanying examples; in Section 3, we will assess the accuracy of the wave-equation solvers provided by Stride, and we will present examples of tomographic reconstructions in 2D and 3D, using both numerical and experimental data; finally, we will present our discussion and proceed to our conclusions in Sections 4 and 5, respectively.

2. Methods

2.1. Software structure

Stride has been designed to address the computational and algorithmic complexity of tomographic imaging by providing high-level interfaces that are modular and extensible, and that closely match the mental framework of domain specialists. It has been implemented in Python, a high-level, interpreted programming language that provides characteristics such as portability, ease of use, and dynamic typing. We have chosen Python because it is the *de facto* language for scientific computing and machine learning, with a large community and package ecosystem.

The high-level interfaces provided by Stride are aimed at addressing five fundamental aspects in high-performance ultrasound computed tomography (Fig. 1):

1. first, abstractions and tools are provided for the solution of optimisation problems, which are the basis for most tomographic imaging algorithms;
2. based on these, a series of classes encapsulate the definition of the tomographic problem being solved, e.g. the transducers employed or the signals used to excite them;
3. the relevant physical processes, such as acoustic or elastic wave propagation, are then modelled by using appropriate solvers that execute high-performance code through DSLs like Devito;
4. scaling of these algorithms, from a local workstation to HPC clusters, is achieved by using an integrated parallelisation library called Mosaic;
5. finally, tools are provided for saving and loading the different components of the problem using a standardised file format.

Each of these will be presented in detail in the following five sections.

2.2. Abstractions for solving optimisation problems

Techniques such as ultrasound computed tomography, optoacoustic tomography [19], or even ultrasound calibration techniques like spatial response identification [20,21], are commonly formulated as mathematical optimisation problems, which are solved numerically by using local methods like gradient descent. Therefore, a fundamental necessity when implementing these techniques is the availability of abstractions that allow us to pose our optimisation problems, calculate gradients of those problems with respect to the relevant parameters, and then apply these gradients through some local optimisation algorithm. In the next paragraphs, we introduce the abstractions that, being at the core of Stride, enable the solution of such inverse problems.

Consider a continuously differentiable function $f(\mathbf{y})$, which can be expressed as $f(\mathbf{y}) = \left\langle \hat{f}(\mathbf{y}), 1 \right\rangle$ with some adequate function $\hat{f}(\mathbf{y})$ and some bilinear form $\langle \alpha, \beta \rangle$. We know that the derivative of $f(\mathbf{y})$ with respect to $\mathbf{y}$ is,

$$\nabla_{\mathbf{y}} f(\mathbf{y}) \delta\mathbf{y} = \left\langle \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}) \delta\mathbf{y}, 1 \right\rangle = \left\langle \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}), \delta\mathbf{y} \right\rangle \tag{1}$$

where $\nabla_{\mathbf{y}} f(\mathbf{y}) \delta\mathbf{y}$ represents the derivative of an operator $f(\mathbf{y})$ in the direction $\delta\mathbf{y}$, and the derivative is by definition linear in the differentiation direction. Consider now that $\mathbf{y} = \mathbf{g}(\mathbf{z})$ is another continuously differentiable function. Then the derivative of $f(\mathbf{y})$ with respect to $\mathbf{z}$ is,

$$\nabla_{\mathbf{z}} f(\mathbf{y}) \delta\mathbf{z} = \left\langle \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}), \delta\mathbf{y} \right\rangle = \left\langle \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}), \nabla_{\mathbf{z}} \mathbf{g}(\mathbf{z}) \delta\mathbf{z} \right\rangle \tag{2}$$

by virtue of the product rule. At this point, we introduce the concept of the adjoint of an operator. Given an operator $D\cdot$, its adjoint is $D^*\cdot$, defined so that $\langle a, Db \rangle = \langle b, D^* a \rangle$. Then, we can rewrite the expression as,

$$\begin{aligned} \nabla_{\mathbf{z}} f(\mathbf{y}) \delta\mathbf{z} &= \left\langle \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}), \nabla_{\mathbf{z}} \mathbf{g}(\mathbf{z}) \delta\mathbf{z} \right\rangle \\ &= \left\langle \nabla_{\mathbf{z}}^* \mathbf{g}(\mathbf{z}) \nabla_{\mathbf{y}} \hat{f}(\mathbf{y}), \delta\mathbf{z} \right\rangle \end{aligned} \tag{3}$$

[1] https://github.com/trustimaging/stride

[2] https://stridecodes.readthedocs.io

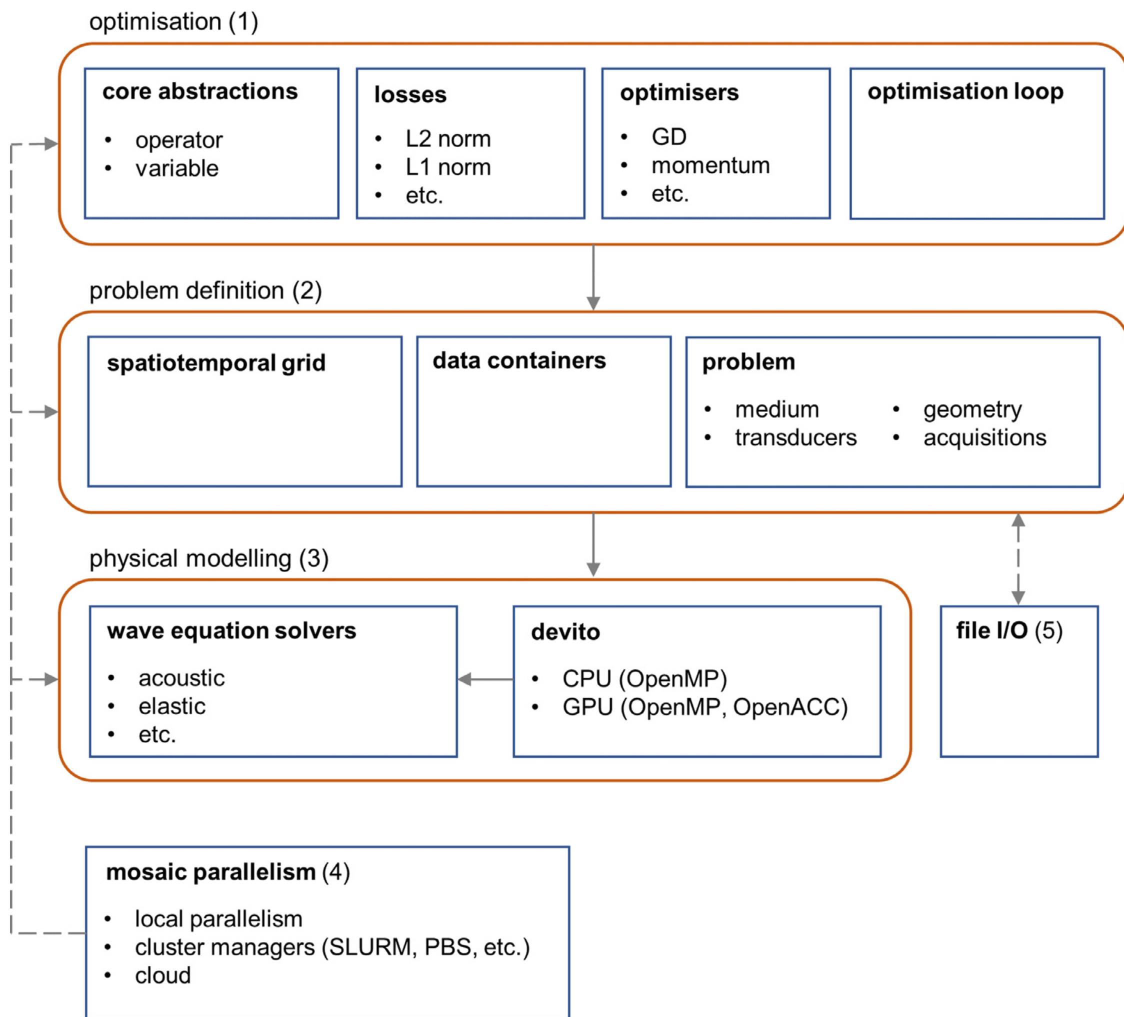


Fig. 1. Schematic representation of the Stride software structure. A series of basic abstractions for solving optimisation problems are provided (1), based on which the tomographic problem is expressed (2). The tomographic problem becomes fully defined when appropriate physical modelling is introduced (3). The execution of Stride is parallelised using the custom library Mosaic (4), and tools are provided to save and load its details (5).

That is, the derivative of function $f(\mathbf{y})$ with respect to $\mathbf{z}$ can be calculated by finding the derivative of $\hat{f}(\mathbf{y})$ with respect to its input $\mathbf{y}$ and then applying the adjoint of the Jacobian of $\mathbf{g}(\mathbf{z})$ on the result. In the discrete case, this is equivalent to the vector-Jacobian product.

Similarly, if we added a third function $\mathbf{z} = \mathbf{h}(\mathbf{x})$, then the same result could be obtained for the derivative of $f(\mathbf{y})$ with respect to $\mathbf{x}$,

$$\begin{aligned} \nabla_{\mathbf{x}} f(\mathbf{y})\delta\mathbf{x} &= \left\langle \nabla^*_{\mathbf{z}}\mathbf{g}(\mathbf{z})\nabla_{\mathbf{y}}\hat{f}(\mathbf{y}), \delta\mathbf{z} \right\rangle \\ &= \left\langle \nabla^*_{\mathbf{z}}\mathbf{g}(\mathbf{z})\nabla_{\mathbf{y}}\hat{f}(\mathbf{y}), \nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x})\delta\mathbf{x} \right\rangle \\ &= \left\langle \nabla^*_{\mathbf{x}}\mathbf{h}(\mathbf{x})\nabla^*_{\mathbf{z}}\mathbf{g}(\mathbf{z})\nabla_{\mathbf{y}}\hat{f}(\mathbf{y}), \delta\mathbf{x} \right\rangle \end{aligned} \quad (4)$$

and the same procedure could be followed for any arbitrary chain of functions for whose inputs we wanted to calculate a derivative. This procedure, known as the adjoint method or backpropagation in the field of machine learning, is effectively the reverse mode that automatic differentiation libraries provide to calculate derivatives, albeit in the continuous limit. This is the core abstraction used in Stride.

Stride considers all components in the optimisation problem, from PDEs to objective functions, as mathematical functions that can be arbitrarily composed, and whose derivative can be automatically calculated through the procedure presented above. In Stride, each of these functions is a `stride.Operator` object, where their inputs and outputs are `stride.Variable` objects (Listing 1).

```
from stride import Variable

x = Variable(name="x",
             needs_grad=True)
z = await h(x)
y = await g(z)
w = await f(y)

await w.adjoint()
# The gradient is now in "x.grad"
```

Listing 1. Example calculation of the gradient of a chain of functions using Stride. Note the use of the `await` syntax that is needed for compatibility with the Mosaic parallelisation library.

When each `stride.Operator` is called, it is immediately applied on its inputs to generate some outputs. At the same time, these outputs keep a record of the chain of calls that have led

to them within a directed acyclic graph. When `w.adjoint()` is called, this graph is traversed from the root `w` to the leaf `x`, calculating the gradient in the process. Only the leaves for which the flag `needs_grad` is set to `True` will have their gradient computed, which will be stored in the internal buffer of the variable `x.grad`.

Now, we proceed to apply these general abstractions to find the gradient of a more practical optimisation problem. Consider the PDE-constrained optimisation problem,

$$\begin{aligned} \mathbf{m}^* = \text{argmin}_{\mathbf{m}} \quad & J(\mathbf{u},\mathbf{m}) = \text{argmin}_{\mathbf{m}} \langle \hat{J}(\mathbf{u},\mathbf{m}), 1\rangle \\ & s.t. \mathbf{L}(\mathbf{u},\mathbf{m}) = \mathbf{0} \end{aligned} \tag{5}$$

given some scalar objective function or loss function $J(\mathbf{u},\mathbf{m})$ and some PDE $\mathbf{L}(\mathbf{u},\mathbf{m}) = \mathbf{0}$, for some vector of state variables $\mathbf{u}$ and a vector of design variables $\mathbf{m}$. Considering $\mathbf{L}(\mathbf{u},\mathbf{m})$ to be an adequate, continuously differentiable function in some neighbourhood of $\mathbf{m}$, we can apply the implicit function theorem. Then $\mathbf{L}(\mathbf{u},\mathbf{m}) = \mathbf{0}$ has a unique continuously differentiable solution $\mathbf{u}(\mathbf{m})$ and its derivative is given by the solution of,

$$\begin{aligned} &\nabla_{\mathbf{u}}\mathbf{L}(\mathbf{u}(\mathbf{m}),\mathbf{m})\nabla_{\mathbf{m}}\mathbf{u}(\mathbf{m})\delta\mathbf{m} + \nabla_{\mathbf{m}}\mathbf{L}(\mathbf{u}(\mathbf{m}),\mathbf{m})\delta\mathbf{m} = \mathbf{0} \\ &\nabla_{\mathbf{m}}\mathbf{u}(\mathbf{m})\delta\mathbf{m} = -\nabla_{\mathbf{u}}\mathbf{L}^{-1}(\mathbf{u}(\mathbf{m}),\mathbf{m})\nabla_{\mathbf{m}}\mathbf{L}(\mathbf{u}(\mathbf{m}),\mathbf{m})\delta\mathbf{m} \end{aligned} \tag{6}$$

We can then define a reduced objective $F(\mathbf{m}) = J(\mathbf{u}(\mathbf{m}),\mathbf{m}) = \langle \hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), 1\rangle$, and we can take its derivative with respect to $\mathbf{m}$ by using the previously introduced procedure,

$$\begin{aligned} \nabla_{\mathbf{m}} \quad F(\mathbf{m})(\delta\mathbf{m}) &= \langle \nabla_{\mathbf{u}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \nabla_{\mathbf{m}}\mathbf{u}(\mathbf{m})\delta\mathbf{m}\rangle \\ &+ \langle \nabla_{\mathbf{m}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &= \langle \nabla^*_{\mathbf{m}}\mathbf{u}(\mathbf{m})\nabla_{\mathbf{u}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &+ \langle \nabla_{\mathbf{m}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \end{aligned} \tag{7}$$

Substituting expression 6 into expression 7 we obtain,

$$\begin{aligned} \nabla_{\mathbf{m}} \quad F(\mathbf{m})(\delta\mathbf{m}) &= \langle \nabla^*_{\mathbf{m}}\mathbf{u}(\mathbf{m})\nabla_{\mathbf{u}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &+ \langle \nabla_{\mathbf{m}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &= -\langle \nabla_{\mathbf{m}}\mathbf{L}^*(\mathbf{u}(\mathbf{m}),\mathbf{m})\nabla_{\mathbf{u}}\mathbf{L}^{-*}(\mathbf{u}(\mathbf{m}),\mathbf{m}) \\ &\nabla_{\mathbf{u}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &+ \langle \nabla_{\mathbf{m}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \\ &= \langle \nabla_{\mathbf{m}}\mathbf{L}^*(\mathbf{u}(\mathbf{m}),\mathbf{m})\mathbf{w}(\mathbf{m}), \delta\mathbf{m}\rangle \\ &+ \langle \nabla_{\mathbf{m}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}), \delta\mathbf{m}\rangle \end{aligned} \tag{8}$$

where $\mathbf{w}(\mathbf{m})$ is the solution of the adjoint PDE,

$$\mathbf{w}(\mathbf{m}) = -\nabla_{\mathbf{u}}\mathbf{L}^{-*}(\mathbf{u}(\mathbf{m}),\mathbf{m})\nabla_{\mathbf{u}}\hat{J}(\mathbf{u}(\mathbf{m}),\mathbf{m}) \tag{9}$$

In this optimisation problem, both $\mathbf{L}(\mathbf{u},\mathbf{m})$ and $J(\mathbf{u},\mathbf{m})$ would be `stride.Operator` objects. Adding new functions to Stride requires defining a new `stride.Operator` subclass that implements two methods, `forward` and `adjoint` (Listing 2).

The abstractions presented allow us to intuitively pose optimisation problems and calculate derivatives of an objective function with respect to the parameters of interest. However, in order to solve the problem, we have to apply this derivative to update our guess of the parameters and repeat the procedure iteratively until we are satisfied with the final result.

Stride provides local optimisers of type `stride.LocalOptimiser` that determine how parameters should be updated given an available derivative. For our previous example, we can follow the procedure in Listing 3 to apply a step of gradient descent in the direction of our calculated derivative. Writing new, user-defined optimisers only requires the creation of a `stride.LocalOptimiser` subclass that takes the `stride.Variable` being optimised when the class is instantiated and that defines the method `step()`, which executes a single step in the optimisation process.

In order to iterate through the optimisation procedure, we could use a standard Python `for` loop. However, we also provide

```python
from stride import Operator, Variable

class L(Operator):
    def forward(self, m):
        # Compute wave equation solution
        return u

    def adjoint(self, grad_u, m):
        # Calculate derivative wrt to m
        # applying adjoint on grad_u
        return grad_m

class J(Operator):
    def forward(self, u, m):
        # Calculate loss value
        return loss

    def adjoint(self, grad_loss, u, m):
        # Calculate the derivative wrt u
        # Calculate the derivative wrt m
        return grad_u, grad_m

# Create the design parameters
m = Variable(name="m")
m.needs_grad = True

# Instantiate the operators
l = L()
j = J()

# Apply to calculate gradient
u = await l(m)
loss = await j(u, m)

await loss.adjoint()
# The gradient is now in "m.grad"
```

Listing 2. Example of gradient calculation for a PDE-constrained optimisation problem like the one solved in FWI.

```python
from stride import GradientDescent

optimiser = GradientDescent(m, step_size=1.)
await optimiser.step()
```

Listing 3. Once a gradient has been calculated, a step in the optimisation algorithm can be taken by using a `stride.LocalOptimiser`.

in Stride a `stride.OptimisationLoop` to use in these cases, which will help structure and keep track of the optimisation process.

Iterations in Stride are grouped together in blocks, with the `stride.OptimisationLoop` containing multiple blocks and each block containing multiple iterations. Partitioning the inversion in this way allows us to divide the optimisation more easily into logical units that share some characteristics. For instance, in FWI it is common to gradually introduce frequency information into the inversion to better condition the optimisation. In this case, it would make sense to assign one block to each frequency band, and run that band for some desired number of iterations. Listing 4 adds a `stride.OptimisationLoop` around our previous example.

```
from stride import OptimisationLoop

opt_loop = OptimisationLoop()

for block in opt_loop.blocks(num_blocks):
    for iteration in \
            block.iterations(num_iters):
        m.clear_grad()

        u = await l(m)
        loss = await j(u, m)
        await loss.adjoint()

        await optimiser.step()
```

Listing 4. Running through multiple iterations in the optimisation can be easily structured using the `stride.OptimisationLoop`.

```
from stride import Space, Time, Grid

space = Space(shape, spacing)
time = Time(start, step, num)

grid = Grid(space, time)
```

Listing 5. Example spatiotemporal grid.

```
from stride import Medium, ScalarField

medium = Medium(grid=grid)
medium.add(ScalarField(name="vp",
                       grid=grid))
medium.add(ScalarField(name="rho",
                       grid=grid))

medium.vp.fill(1500.)
medium.rho.fill(1000.)
```

Listing 6. Example `stride.Medium` containing the spatial distribution of longitudinal speed of sound and density.

2.3. Problem definition

In addition to providing abstractions for solving optimisation problems, Stride introduces a series of utilities for users to specify the characteristics of the problem being solved, such as the physical properties of the medium or the sequence in which transducers are used.

In Stride, the problem is first defined over a spatiotemporal grid, which determines the spatial and temporal bounds of the problem and their discretisation (Listing 5). Currently, we support discretisations over rectangular grids, but other types of meshes could be introduced in the future. On this spatiotemporal mesh, we define a series of grid-aware data containers, which include scalar and vector fields, and time traces. These data containers are subclasses of `stride.Variable`.

Based on this, we can define a medium, a `stride.Medium` object, a collection of fields that determine the physical properties in the region of interest. For instance, the medium could be defined by two `stride.ScalarField` objects containing the spatial distribution of longitudinal speed of sound and density, as in Listing 6.

```
from stride import PointTransducer, \
                   Transducers, Geometry

transducers = Transducers(grid=grid)
trans_0 = PointTransducer(0, grid=grid)
trans_1 = PointTransducer(1, grid=grid)

transducers.add(trans_0)
transducers.add(trans_1)

geometry = Geometry(transducers=transducers,
                    grid=grid)
geometry.add(0,
             transducer=trans_0,
             coordinates=[...])
geometry.add(1,
             transducer=trans_1,
             coordinates=[...])
```

Listing 7. Example geometry with its associated transducers.

```
from stride import Shot, Acquisitions

loc_0 = geometry.get(0)
loc_1 = geometry.get(1)

acquisitions = Acquisitions(geometry=geometry,
                            grid=grid)
shot = Shot(0,
            sources=[loc_0],
            receivers=[loc_0, loc_1],
            geometry=geometry,
            grid=grid)
acquisitions.add(shot)
```

Listing 8. Example acquisition containing only one shot.

Next, we can define the transducers, the computational representation of the physical devices that are used to emit and receive sound, characterised by aspects such as their geometry and impulse response. These transducers are then located within the spatial grid by defining a series of locations in a `stride.Geometry`. In Listing 7 we instantiate some `stride.Transducer` objects and then add them to a corresponding `stride.Geometry`.

Finally, we can specify an acquisition sequence within a `stride.Acquisitions` object (Listing 8). The acquisition sequence is composed of shots (`stride.Shot` objects), where each shot determines which transducers at which locations act as sources and/or receivers at any given time during the acquisition process. The shots also contain information about the wavelets used to excite the sources and the data observed by the corresponding receivers if this information is available.

All components of the problem definition can be stored in a `stride.Problem` object, which structures them in a single, common entity.

2.4. Physical modelling

Physical modelling is defined in Stride through `stride.Operator` objects that represent specific implementations of a numerical solver applied to a PDE. Stride does not prescribe a specific solver or numerical method, and different codes and implementations can be integrated with it as long as they conform to the `stride.Operator` interface.

By default, Stride integrates with the Devito library, a domain-specific language that generates highly optimised finite-difference code from high-level symbolic differential equations [17,18]. Using Devito, we provide an out-of-the-box implementation of the second-order isotropic acoustic wave equation, for which Devito automatically generates code that can be readily executed in parallel on CPUs using Open Multi-Processing (OpenMP), and on GPUs using both OpenMP and OpenACC.

Acoustic modelling in Stride is governed by the equation,

$$\frac{1}{v_p^2}\frac{\partial^2 p}{\partial t^2} = \rho \nabla \cdot \left(\frac{1}{\rho}\nabla p\right) + \eta \frac{\partial}{\partial t}(-\nabla^2)^{y/2} p \tag{10}$$

where $p(t, \mathbf{x})$ is the pressure, $v_p(\mathbf{x})$ is the longitudinal speed of sound, $\rho(\mathbf{x})$ is the mass density, $\eta = -2\alpha_0 v_p^{y-1}$, and $\alpha_0(\mathbf{x})$ is the absorption coefficient. The implementation of the acoustic wave equation is fourth-order accurate in time and tenth-order accurate in space. This results in a stability region with Courant-Friedrichs-Lewy (CFL) constant of 0.80 in 2D and 0.66 in 3D [22], as well as the requirement of a minimum of 3 points per wavelength (PPW) to minimise numerical dispersion. Our solver includes options for both constant and variable density and attenuation. Attenuation follows a power law, with frequency dependence controlled by the parameter y in the equation, which can take values 0 and 2. In these cases the implemented derivative is not fractional.

In terms of boundary conditions, Stride includes options for a sponge absorbing boundary [23] or a perfectly matched layer [24]. In all cases, sources and receivers can be defined in locations off the grid, with both bi-/tri-linear interpolation and high-order sinc interpolation [25]. It is important to note that current, out-of-the-box implementations of the adjoints of our PDE solvers consider domains to be unbounded, as these represent the most common scenario in ultrasound imaging. However, alternative boundary conditions can be readily accounted for through user-level extensions of the PDE operators.

Although physical modelling in Stride is currently focused on finite-difference methods, future releases could include integration with pseudospectral-element DSLs such as Dedalus [26] or finite-element DSLs like FEniCS/Firedrake [27,28].

2.5. Parallelism

In practice, derivatives of the optimisation problem are not calculated one data point at a time, but in batches, and the result is averaged to obtain an estimate of the gradient for that iteration. Because, in most cases, each of these data points is fully independent, this can be exploited so that they are calculated in parallel. For some simple problems, this can be done within a single workstation. However, in most practical problems, compute and memory demands require that these computations are mapped across different interconnected sets of hardware, such as multi-GPU systems and CPU clusters, running locally, remotely, or on the cloud.

The most important limiting factor when scaling real-life FWI workloads in parallel environments is memory allocation, management, and communication, with potentially hundreds of gigabytes being stored and transferred during the optimisation process. Therefore, a parallelisation framework is required that offers fine-grained control of the computational workload allocation and memory management for code developers, while also providing the end user with a high level of abstraction that integrates tightly with the optimisation constructs provided by Stride. We have developed Mosaic to facilitate the expression of parallelism in Stride in an intuitive manner.

Mosaic is an actor-based parallelisation library based on asynchronous, zero-copy message passing through ZeroMQ sockets [29]. Actors in Mosaic are called tessera, and can be generated by

```python
from mosaic import tessera

@tessera
class Remote:
    def __init__(self):
        self.value = 0
    def add(self, value):
        self.value += value
        return self.value

# Create a remote instance
remote_obj = Remote.remote()

# Check the current value of the attribute
print(await remote_obj.value)

# Add a new value
task = remote_obj.add(5)

# This will return immediately and
# we can do other work in the meantime

# When ready, we can wait for the
# remote method call to finish
await task

# The return value of the method call
# is stored in the remote worker, and
# to access them we will need to do
# this explicitly
print(await task.result())

# Check the new value of the attribute
print(await remote_obj.value)
```

Listing 9. Basic usage of Mosaic to create remote objects, call their methods, and access their attributes.

decorating any Python class using `@mosaic.tessera`. When instantiating a class that has been decorated, Mosaic will start a remote instance of that class in one of the workers. At this point, remote method calls to that tessera can be executed and the attributes of that remote object can be accessed. An example of how Mosaic is used can be found in Listing 9.

In Mosaic, subsequent method calls to a remote object are guaranteed to be executed in order, but calls to different remote objects are not. However, if there are explicit dependencies between two or more remote method calls, Mosaic will ensure that these are executed in the right order (Listing 10).

The structure of the Mosaic runtime, which can be seen in Fig. 2, is composed by a series of processing units, which could be located in a single, local workstation or distributed across a remote network. The first of such units contains a *monitor* process, a *warehouse* process, and a *head* process. The *monitor* process collects information about the Mosaic network, including occupation rate, resource use and connection state. The *warehouse* process acts as a centralised key-value storage location that is accessible from across the whole Mosaic network. The *head* process is the place where the main user code is executed. In each of the remaining processing units, a *node monitor* and one or more *workers* are allocated. The *node monitor* keeps track of the runtime status of its local processing unit and oversees the life cycle of each of the *workers* in its unit. Finally, the *workers* act as containers for tessera actors, whose methods can be executed remotely. All processing units in the Mo-

```
# Create a remote instances
remote_obj_0 = Remote.remote()
remote_obj_1 = Remote.remote()

# These could be executed in parallel
task_0 = remote_obj_0.add(5)
task_1 = remote_obj_1.add(1)

# and have to be awaited separately
await task_0
await task_1

# An explicit dependence will make
# them execute in series
task_0 = remote_obj_0.add(1)
task_1 = remote_obj_1.add(task_0)

# and only the latter needs to be awaited
print(await task_1.result())  # will print 7

# Dependencies can also be introduced as
task_0 = remote_obj_0.add(1)
task_1 = remote_obj_1.add(task_0.done, 2)

await task_1
```

Listing 10. Expressing parallelism and dependencies in Mosaic.

saic network are directly interconnected to each other, creating a decentralised communication mesh.

Mosaic can be run in interactive mode in a Jupyter notebook, or from a terminal window using the `mrun` command. The Mosaic runtime can be used without any code changes in a local multi-processing environment or a multi-node cluster. Therefore, Mosaic gives us the flexibility to parallelise work across multiple CPUs within a single compute node, as well as across multiple interconnected nodes, with the distribution topology related to the specific application at hand. Additionally, our Devito solvers can parallelise the execution of the wave equation across multiple CPU cores by using thread-level parallelism.

2.6. File input and output

As the popularity of ultrasound tomography increases, the number and size of datasets are also growing, but no standard format exists for their exchange. This slows algorithm development and limits research reproducibility. In order to address this, we have introduced with Stride a standardised file specification and a set of tools to interact with it.

In the setup of ultrasound tomography workflows, there are usually a number of intermediate files that are generated describing aspects such as medium properties, transducer impulse responses or data recorded during laboratory experiments. In Stride, we use the Hierarchical Data Format (HDF5) [30] for saving and loading these datasets and provide a series of tools to conveniently interact with them. Figure 3 shows the basic file specification proposed in Stride for the different components of a standard tomographic workflow.

3. Results

3.1. Modelling accuracy

We have validated the accuracy of the acoustic solver by comparing it against an analytical solution of the wave equation for

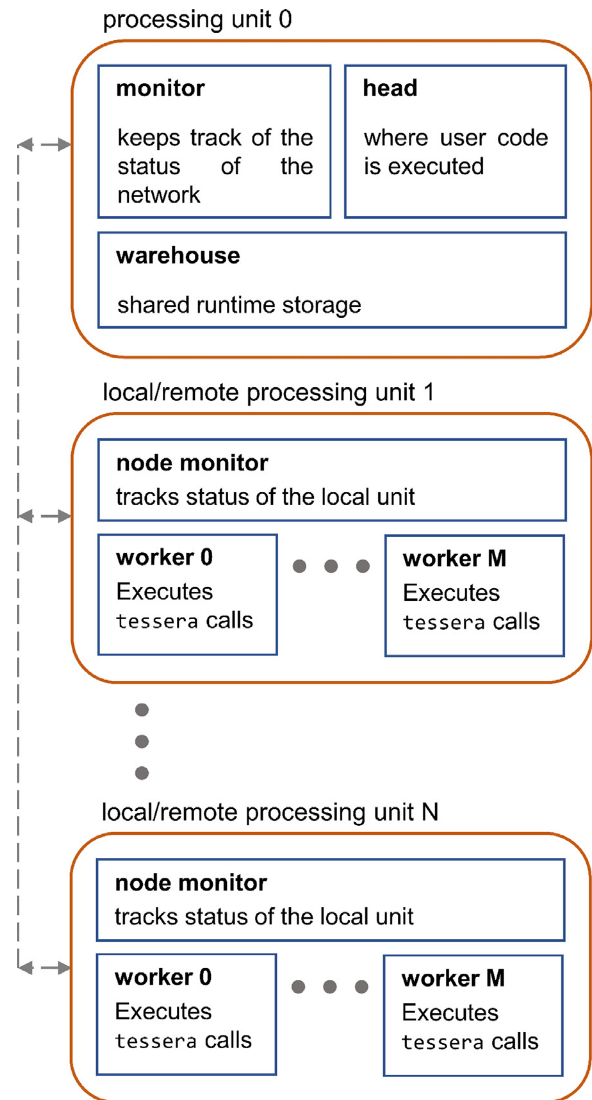


Fig. 2. Schematic representation of the Mosaic runtime. The runtime is divided into several logical processing units, which could represent, for instance, processes in a local multi-processing environment or different machines in a multi-node cluster. In the first processing unit, the user code is executed in the *head*, while the *monitor* tracks the status of the runtime and the *warehouse* acts as a central storage unit. In the remaining processing units, a *node monitor* is allocated to track the status of that local unit and communicate this to the global *monitor*, and one or more *workers* are also created to execute `tessera` calls. All endpoints in the Mosaic runtime are interconnected to each other.

a homogeneous medium [31]. The comparison was performed, in 2D and 3D, by transmitting a three-cycle tone burst centred at 500 kHz into a medium with constant speed of sound of 1500 m/s, constant density, and no attenuation. The employed grid was sampled at 0.250 mm in space (minimum of 8 PPW) and 0.060 μs in time (maximum CFL constant of 0.36). The resulting acoustic wave was then recorded at 51 equispaced points, starting at the transmission location and increasing in distance up to a maximum separation of 300 mm.

Results for the comparison are shown in Fig. 4, both for the 2D (Fig. 4-A) and the 3D cases (Fig. 4-B), where errors with respect to the analytical solution were calculated using the normalised root-mean-square error. We can see how the Stride numerical solutions closely match the analytical ones, remaining accurate at a significant distance from the transmission site.

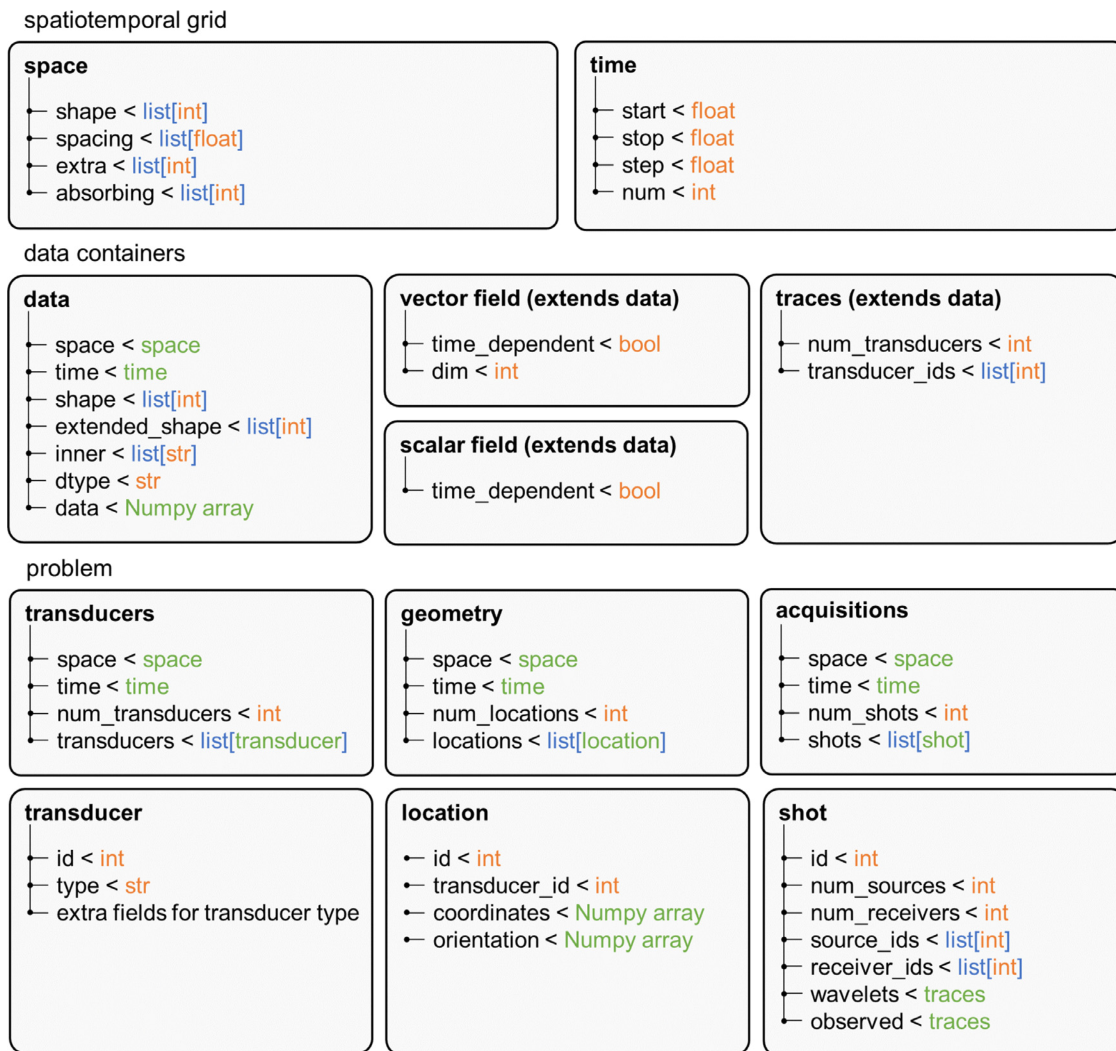


Fig. 3. Specification of the Stride file format. The definition of the spatiotemporal grid is the basis upon which different types of data containers and the various components of the problem are then specified.

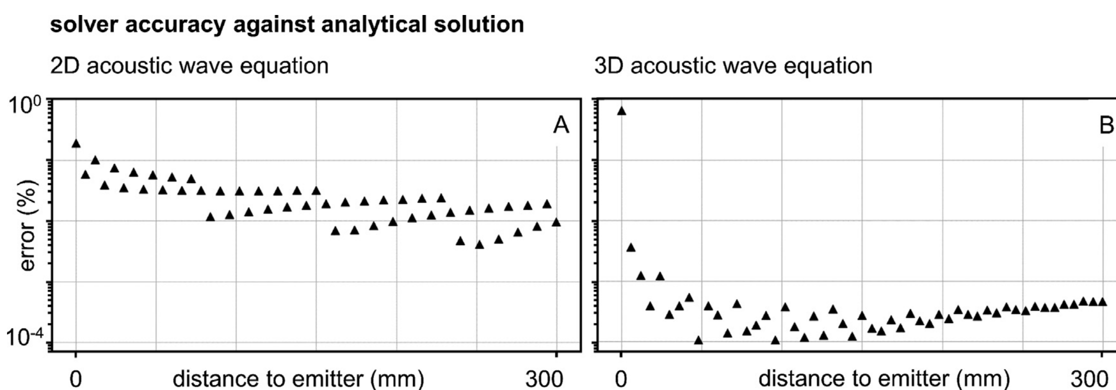


Fig. 4. Accuracy of the acoustic wave equation solver against analytical solution. The numerical solution of the acoustic wave equation calculated by Stride is compared to the analytical solution for a medium with homogeneous speed of sound. The comparison is performed in 2D (A) and 3D (B), at a distance to the emitter ranging from 0 to 300 mm. Error is calculated as the normalised root-mean-square error with respect to the analytical solution.

We have performed a further validation of the Stride acoustic solvers on a more complex medium with inhomogeneous speed of sound, density, and attenuation of order zero, for which an analytical solution is not available, by comparing it against kWave [32], a state-of-the-art ultrasound modelling library written in MATLAB and based on pseudospectral element methods. The comparison was performed using a human skull section, seen in Fig. 5-A, sampled at 0.125 mm (minimum of 24 PPW), and illuminated by a bowl ultrasound transducer with a 64 mm radius of curvature and a 64 mm aperture diameter. The transducer surface was discretised using 20,000 point sources, evenly distributed using Fibonacci spirals [33]. This example forms part of a transcranial ultrasound simulation benchmarking and intercomparison exercise organised by the ITRUSST (International Transcranial Ultrasonic Stimulation

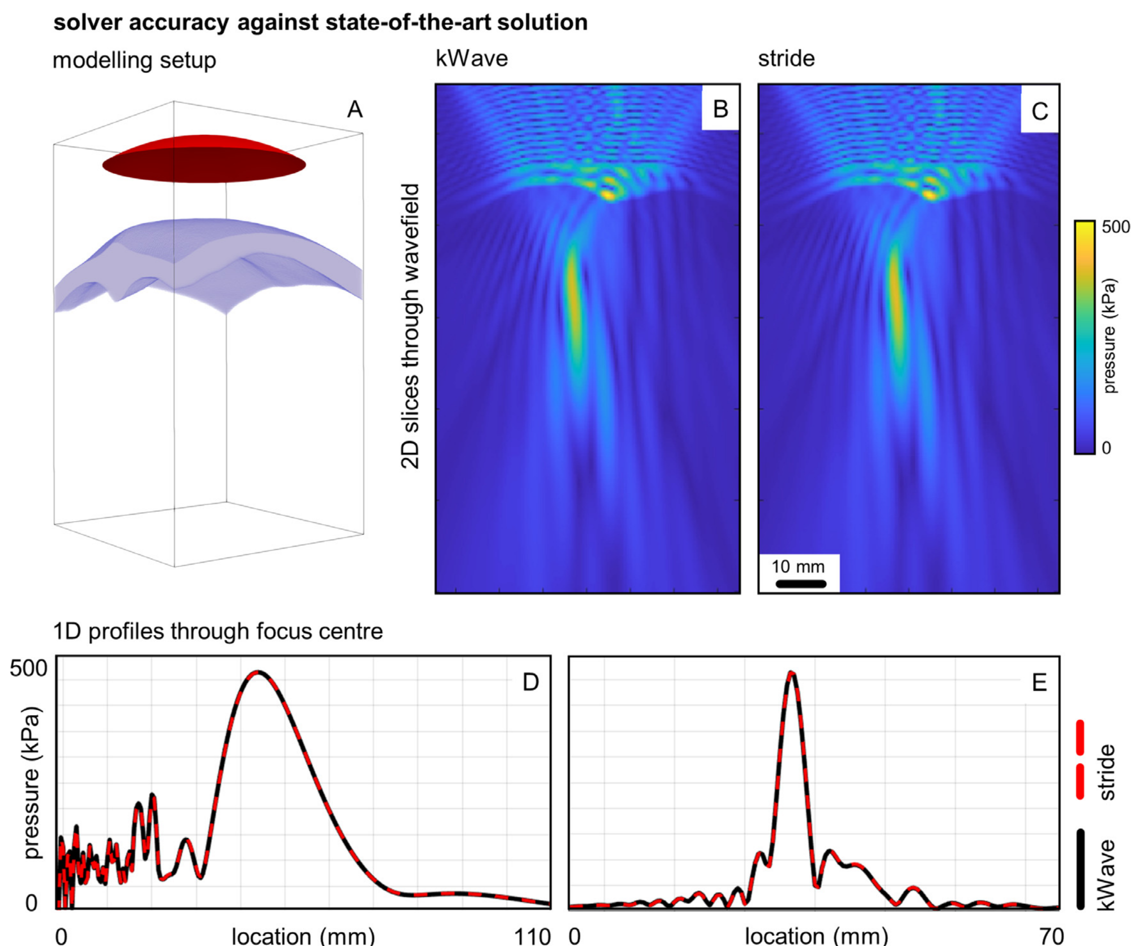


Fig. 5. Accuracy of the acoustic wave equation solver against state-of-the-art solver. The 3D numerical model (A) contains a human skull section (blue) and a bowl ultrasound transducer (red). We compare a 2D slice through the resulting steady-state wavefield for the state-of-the-art solver kWave (B) and for Stride (C). Additionally, we compare two 1D profiles through the centre of the transducer focus (D-E).

Safety and Standards) planning group [34]. The transducer was excited by a continuous sinusoidal wave at 500 kHz and the simulation was run with a step size of 0.016 μs (maximum CFL constant of 0.36) until steady state was reached. The magnitude of the pressure field at the excitation frequency was then extracted after Fourier transform. Fig. 5-B and C show a 2D slice through the resulting 3D wavefield, from which we can observe the good agreement between both solutions. A similar conclusion can be extracted from the 1D profiles, seen in Fig. 5-D and E. The agreement between both solvers is quantitatively confirmed by a relative error of 1.64%, calculated over the entire 3D volume. Existing differences between the results of both solvers are likely due to the use of different numerical methods to solve the wave equation, as well as differences in source injection routines and boundary condition implementation. It is important to note that implementation differences cannot be fully eliminated, even in the limit where both numerical methods converge, due to the fact that Stride and kWave are solving fundamentally different equations in order to model acoustic wave propagation: Stride solves the second-order linear acoustic wave equation, whereas kWave solves three coupled equations that are equivalent to a generalized Westervelt equation.

3.2. Imaging in 2D

For our first imaging experiment, we extract a 2D slice from a numerical breast model as seen in Fig. 6-A. The resulting 2D model can be seen in Fig. 7-A. The model has been obtained from an open database [35], and has been adapted by populating it with acoustic tissue properties and by adding a tumour. From here onwards, all examples were run with constant density and no attenuation. The model, sampled with a spacing of 0.500 mm (minimum of 3.73 PPW), has a size of 456×485. The model is surrounded by 128 point transducers, seen as blue dots in Fig. 6-A, all of which act as sources and receivers. Imaging is performed using a three-cycle tone burst centred at 500 kHz, and is carried out over 200 μs in steps of 0.080 μs (maximum CFL constant of 0.26). Both temporal and spatial sampling are kept constant during modelling and inversion, for this and all subsequent examples. However, this is not required and users could exploit different dispersion and stability conditions by changing the discretisation across different imaging blocks.

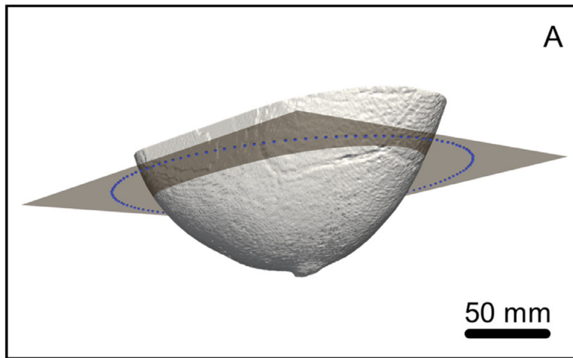
To make use of the gradient-calculation capabilities of Stride, we instantiate our speed-of-sound field with `needs_grad=True`, and set the starting model to a constant sound speed of 1500 m/s (Fig. 7-B). We also instantiate a gradient descent optimiser to update our variable (Listing 11).

We can see in Listing 11 how the `stride.ScalarField` has been instantiated by calling `parameter()`. Using this method will ensure that, as the field is sent across the Mosaic network, a reference to the original object will always be maintained. This will allow us to calculate the gradient in different workers and then propagate the results back to the local runtime.

Then, we can instantiate our operators remotely, creating one copy for each available worker (Listing 12). In this case, we use

imaging setup

2D imaging geometry



3D imaging geometry

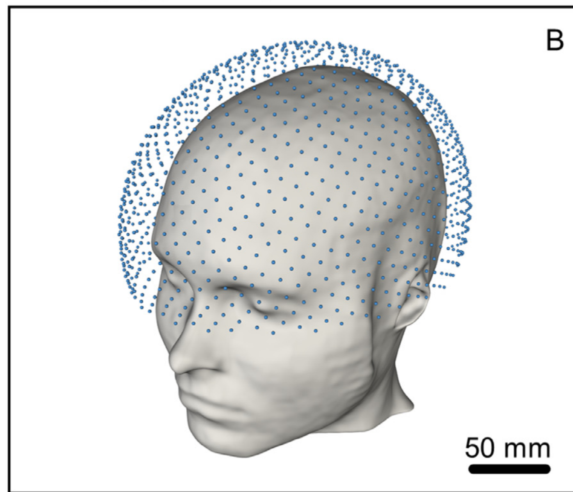


Fig. 6. Setup used in the numerical experiments. For the 2D experiment (A), a slice is taken across a numerical 3D model of the breast and 128 point transducers, which can be seen as blue dots, are distributed around it. For the 3D experiment (B), a numerical head model is imaged by surrounding it with 1024 transducers (also visible as blue dots). The scale shown at the bottom of the numerical models applies equally in all spatial directions.

an operator for the PDE and another one for the objective function, and we also create pre-processing operators for our source wavelets and our output time traces.

We perform the inversion by gradually introducing frequencies, starting at 300 kHz and going up to 600 kHz. We do this by running the optimisation loop in blocks, with each block using a different frequency band. At each block, we complete 8 iterations, randomly selecting 16 shots without replacement in each of them. That is, each shot is used once at every frequency band. We run the function in Listing 13 for every iteration of the reconstruction loop in Listing 14. We run this inversion on a local multi-processing environment, within a Jupyter notebook, by simply adding the command `mosaic.interactive(‘‘on’’)` at the beginning of our notebook. This workstation is equipped with 64 GB of memory and 6 physical cores (Intel i7-8700K, 6 cores, 3.70 GHz). The acoustic Devito PDE was compiled using the GNU `gcc` compiler version 7.5, and was executed on the Jupyter notebook using 3 Mosaic workers and OpenMP thread-level parallelism with 2 threads for each worker. Each of the Mosaic workers calculates the gradient for a single shot at a time, which entails one forward propagation and one adjoint propagation of the acoustic solver, before combining the gradients for all shots at each itera-

2D numerical imaging results

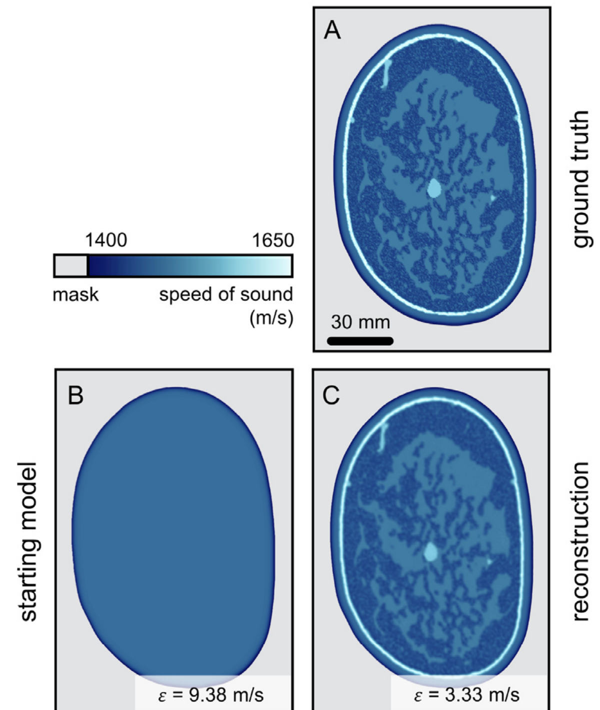


Fig. 7. Stride reconstruction in 2D. A 2D acoustic breast model (A) is imaged starting from a homogeneous distribution of speed of sound (B). Stride manages to accurately reconstruct the target model (C). The mean of the absolute value of the difference between the ground-truth model and the inversion is displayed here as ε.

```
# Prepare starting model
vp = ScalarField.parameter(name="vp",
                           grid=grid,
                           needs_grad=True)

vp.fill(1500.)
medium.add(vp)

# Prepare optimiser
optimiser = GradientDescent(vp,
                            step_size=step_size)
```

Listing 11. To image the spatial distribution of speed of sound, we create a `stride.ScalarField(..., needs_grad=True)` and set the starting distribution to be 1500 m/s everywhere. We also create a `stride.GradientDescent` optimiser to update the variable at every iteration.

```
# Prepare operators
pde = IsoAcousticDevito.remote(grid=grid,
                               len=num_workers)
loss = L2DistanceLoss.remote(
                               len=num_workers)
p_wavelets = ProcessWavelets.remote(
                               len=num_workers)
p_traces = ProcessTraces.remote(
                               len=num_workers)
```

Listing 12. We create the necessary operators for the reconstruction. The keyword argument `len=num_workers` controls the amount of copies of the operators to be instantiated by Mosaic in each remote worker.

```
async def run_iter(f_max):
    # Select some shots for this iteration
    shot_ids = acquisitions.select_shot_ids(
                    num=shots_per_iter,
                    randomly=True)

    # Clear the gradient
    vp.clear_grad()

    # Async loop over selected shots
    @runtime.async_for(shot_ids)
    async def loop(worker, shot_id):
        # Fetch one data point
        sub_problem = problem.sub_problem(
                                  shot_id)
        wavelets = sub_problem.shot.wavelets
        observed = sub_problem.shot.observed

        # Pre-process the wavelets
        wavelets = p_wavelets(wavelets,
                           runtime=worker)
        # Execute the PDE
        modelled = pde(wavelets,
                       vp,
                       problem=sub_problem,
                       runtime=worker)

        # Pre-process traces
        traces = p_traces(modelled,
                          observed,
                          f_max=f_max,
                          runtime=worker)
        # Calculate loss
        fun = await loss(traces.outputs[0],
                         traces.outputs[1],
                         problem=sub_problem,
                         runtime=worker).result()

        # Calculate derivative
        await fun.adjoint()

    # Wait for loop to end
    await loop
    # Update vp
    await optimiser.step()
```

Listing 13. At every iteration, a subset of the available shots are selected randomly to calculate a gradient. The calculated gradient is then used to update the speed of sound distribution.

```
opt_loop = OptimisationLoop()

# Start optimisation
for block, f_max in \
        opt_loop.blocks(num_blocks, freqs):
    # Every iteration in the block
    for iteration in \
            block.iterations(num_iters):
        await run_iter(f_max)
```

Listing 14. The inversion is performed by selecting subsequent frequency bands and, in each band, a certain number of iterations are run to calculate a gradient.

tion. With this configuration, each shot gradient calculation took a total of 2.99 $\pm$ 0.30 s.

Once the optimisation loop runs through all frequency bands, a final reconstruction is obtained (Fig. 7-C). We calculate the mean of the absolute value of the difference between the final reconstruction and the original model, which is displayed in Fig. 7 with the symbol ε. We can see how the reconstruction closely matches the ground-truth model, both qualitatively and quantitatively. As expected, inaccuracies can be observed in the reconstruction, which can be explained through a number of factors. First, limited sampling of the wavefield is performed at the boundaries of the model because a finite number of receivers is used. Second, the available frequency bandwidth is also necessarily finite, which will limit resolution and prevent high-contrast interfaces from being perfectly recovered.

Next, we apply the same imaging script that we have just introduced to now image an experimental tissue-mimicking phantom. A polyvinyl alcohol (PVA) cryogel phantom was constructed with two layers of different speed of sound values and an inner cavity filled with water [36]. The dimensions of the phantom are, approximately, 57.4 mm in width, 70.4 mm in height, and 130 mm in depth. Speed-of-sound values for the different layers of the phantom were experimentally measured using time of flight to be 1521 $\pm$ 3 m/s for the outer layer and 1502 $\pm$ 4 m/s for the inner layer. A photograph of the cross section of the phantom can can be seen in Fig. 8-A. Data were then acquired using two P4-1 transducers (ATL, USA), each of which contains 96 transmitting and receiving elements. The two P4-1 transducers were independently attached to two rotary motors, allowing them to move around the phantom for full illumination. Data were acquired by transmitting with a centre frequency of 1.4 MHz.

The inversion was performed over 120 μs, in steps of 0.048 μs (maximum CFL constant of 0.37), using a spatial sampling of 0.200 mm (minimum of 4.67 PPW) and a grid size of 890$\times$890. Imaging was carried out using a single block and a single frequency band with an upper limit of 700 kHz across a total of 152 iterations. During each iteration, 10 shots were selected randomly without replacement so that each shot was used four times at the end of the block. A single frequency band is sufficient in this example because, for this particular experiment, the starting point of our inversion is close enough to the minimum of the optimisation to ensure convergence. Simultaneously, the resolution offered by this frequency band (with a half-wavelength of approximately 1 mm in water) is sufficient, given the size and level of detail of the phantom, to recover a high-resolution reconstruction.

Using a starting model that contained homogeneous water (Fig. 8-B), a high-resolution reconstruction of the phantom is obtained (Fig. 8-C). Stride can successfully recover the two layers of speed of sound, as well as the internal water cavity. The reconstruction shows high contrast between layers, and the correct recovery of the complex details at the interface between them. We can also see how, at some points, the two layers of the phantom seem to gradually dissolve into one another instead of presenting sharp interfaces. This could be an imaging artefact due to errors in the calibration of the data acquisition setup, but could also be due to the natural degradation of the phantom, which could have led to the two layers merging at these locations.

We run this inversion on the same workstation as the previous example, using the same 3 Mosaic workers, so that each shot gradient calculation took a total of 28.32 $\pm$ 4.46 s. Adding a single argument to the PDE call, `pde(..., platform=''nvidia-acc'')`, is sufficient to run the same inversion on an available GPU instead of the CPU. In this case, the Devito-generated OpenACC solver is compiled using the PGI `pgc++` compiler version 21.2. Then, using the same workstation, equipped with an NVIDIA GeForce RTX 2080 Ti with 11 GB of memory, and

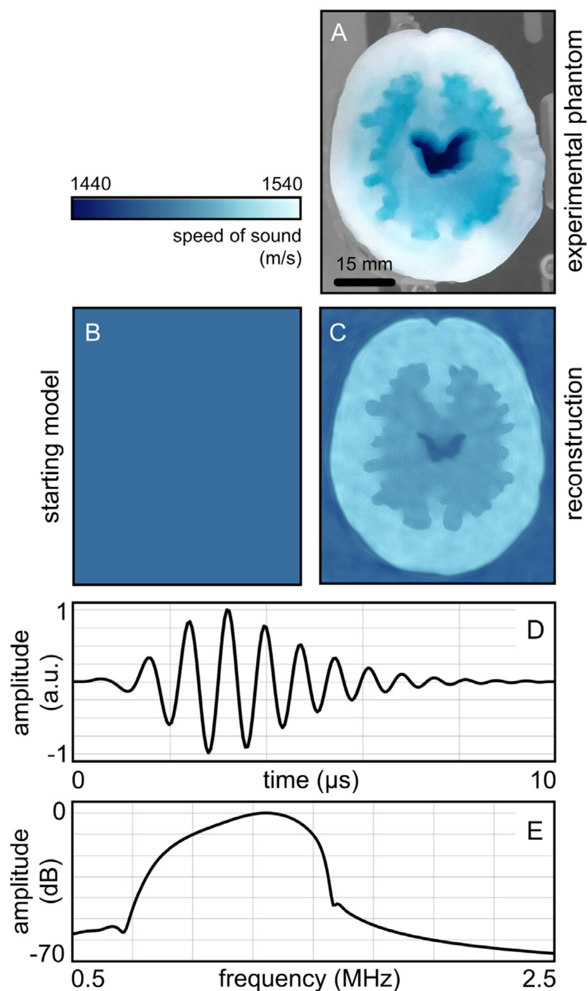


Fig. 8. Experimental Stride reconstruction in 2D. A tissue-mimicking phantom (A) is imaged starting from a homogeneous distribution of speed of sound (B). The Stride reconstruction (C) closely matches the target phantom, is able to recover the different layers of speed of sound and the complex interface between those layers. We can also see the signal used experimentally for imaging (D) and its corresponding magnitude spectrum (E).

a single Mosaic worker, each shot gradient calculation took a total of 5.63 ± 0.07 s.

3.3. Imaging in 3D

Although relevant when imaging structurally simple, soft tissues such as the breast, 2D imaging on its own is of limited applicability in realistic tomographic reconstructions, where 3D modelling and inversion is needed to account for the full physics of wave propagation in the human body. At the same time, it is in these 3D problems where the computational cost of FWI is most apparent and where tomography codes are required to scale robustly. In order to showcase the scaling capabilities of Stride, we choose for our second experiment a numerical 3D model of the adult human head (Fig. 6-B). The model is based on the MIDA model [37], to which acoustic properties were assigned as described by Guasch et al. [4]. Three slices through this numerical model can be seen in Fig. 9-A to C. The model is sampled with a spacing of 0.750 mm (minimum of 3.22 PPW), resulting in a grid of size 367×411×340 and more than 51 million unknown parameters to be estimated. A total of 1024 transducers were located around the head as seen in Fig. 6-B, with all transducers acting both as sources and receivers. Imaging was performed with a three-cycle tone burst centred at 500 kHz. Modelling was carried out over 300 μs, with time steps of 0.150 μs (maximum CFL constant of 0.60).

Stride has been designed to seamlessly scale from 2D to 3D, and moving from one to the other only requires changing three lines of the code when defining the spatial grid. The remaining code can be run without any changes. In this case, the reconstruction is performed in the frequency range between 100 kHz and 600 kHz, starting from a model that only contains the skull (Fig. 9-D to F). Each frequency band in the reconstruction is run for 8 iterations, and 128 shots are randomly selected without replacement for each of them.

Due to the higher computational requirements in 3D, we run this reconstruction in an HPC cluster environment. Except for removing the `mosaic.interactive(''on'')` command, no changes are required to the code when scaling from the local to the cluster environment. Each compute node in the cluster is equipped with 256 GB of memory and 128 cores (2xAMD Zen2 EPYC 7742, 64 cores, 2.25 Ghz). Nodes are connected using an HPE Slingshot interconnect with 200 Gb/s signalling. The Devito solver is compiled using the GNU `gcc` compiler version 7.5, and is executed using OpenMP thread-level parallelism across 32 threads.

Each of the nodes calculates the gradient for a single shot at a time, which once more entails one forward propagation and one adjoint propagation of the acoustic solver, before combining the gradients for all shots at each iteration. Work distribution across the different nodes is managed by the Mosaic runtime, with the time taken to allocate this work generally dominated by the serialisation, communication, and processing of the data associated with the execution of each shot. However, serialisation in Mosaic has a negligible impact due to its zero-copy implementation. Communication overheads could have an impact on performance, but these are minimised by high-speed interconnects and by the asynchronous nature of Mosaic and its underlying ZeroMQ sockets. This means that user code is not slowed down by the actual time taken to send messages across the network by allowing the overlap of computation and communication: the *head* process dispatches all shots almost instantaneously, and independent *worker* processes across the network start computing as soon as the first message arrives. Message processing, on the contrary, will have an impact on performance due to the intrinsic single-threaded nature of Python. This could be alleviated by offloading some of this processing to lower-abstraction interfaces in C. With all this in mind, each shot gradient calculation took 5.82 ± 0.36 min, including time spent in work distribution.

The high accuracy of the final reconstruction obtained using Stride can be seen in Fig. 9-G to I. Also in this case, we have calculated a corresponding quantitative error measure for the full 3D model, shown in Fig. 9 with the symbol ε. Errors in the reconstruction can in this case be attributed to similar reasons to the previous numerical 2D case, with the added factor of limited illumination in certain regions of the model. We can see, for example, how the regions close to the neck and around the sinuses are more poorly resolved due to the location of sources and receivers around the head. We can also see how resolution is degraded as we move closer to the upper regions of the skull due to lower ray density in these areas.

At this point, we explore the scaling capabilities of the Mosaic parallelisation layer by running a fixed number of individual shot gradient calculations, 128, while increasing the number of compute nodes used in the HPC cluster. The achieved acceleration is calculated by comparing the amount of time taken to complete all gradient calculations using a certain number of nodes with respect to the time taken using a single node. Under ideal circumstances, this means that, for example, an acceleration of 128 times is expected when using 128 compute nodes. This test is repeated five times,

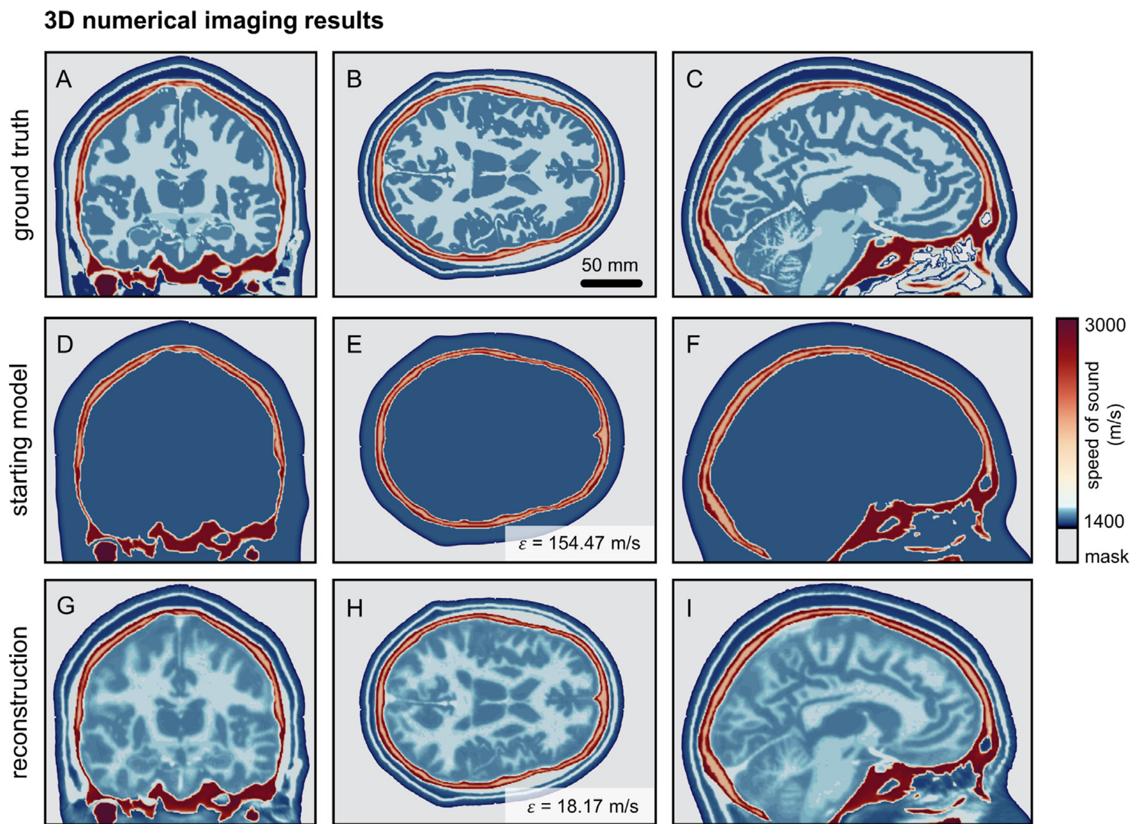


Fig. 9. Stride reconstruction in 3D. A 3D acoustic head model (top row) is imaged starting from a model that contains only the skull and is homogeneous otherwise (middle row). Stride manages to accurately reconstruct the target model (bottom row). The mean of the absolute value of the difference between the ground-truth model and the inversion is displayed here as ε.

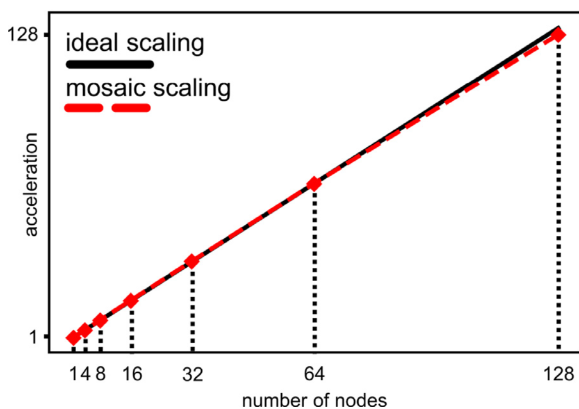


Fig. 10. Mosaic strong scaling for the 3D head model. Scaling obtained with Mosaic (red, dashed line) is compared to the ideal scaling scenario (black, continuous line). Scaling is analysed by running 128 shot gradient calculations for the 3D head model across an increasing number of compute nodes. Acceleration is calculated as the amount of time taken to complete all gradient calculations using a certain number of nodes with respect to the time taken using a single node, averaged over 5 experiments.

and the final acceleration is taken as the average over all repetitions.

Results for this strong scaling test can be seen in Fig. 10, where we can observe that Mosaic achieves nearly ideal scaling up to 128 compute nodes. For the largest number of nodes, we can see how the obtained acceleration deviates slightly from the ideal curve. This corresponds, approximately, to a 2% loss in performance, which can be attributed to the effective single-threaded nature of Python programs that we have previously discussed.

4. Discussion

We have shown that Stride provides an intuitive framework for the solution of ultrasound tomography problems, seamlessly switching between 2D and 3D applications, and between a local workstation and a multi-node cluster.

Implementations of ultrasound tomography methods like FWI have to address their computational and algorithmic complexity. To do this, Stride has been designed to provide tailored optimisation routines, high-performance PDE solvers, and scalability to HPC systems, while simultaneously offering a high level of abstraction to ensure flexibility, productivity, and modularity.

From the point of view of the optimisation, we have seen how Stride closely matches the mathematical formulation of the inverse problem, for which gradients can be intuitively calculated using the adjoint method. Our approach here resembles that taken by machine learning libraries like PyTorch [38], which have been highly successful at broadening the reach of these technologies beyond computational experts. This serves the double purpose of easing adoption by users, some of which might already be familiar with some of these libraries, and facilitating integration with these machine learning tools.

We have to note that gradients for Stride problems are calculated at a high level by treating the PDE or the loss functions as differentiable primitives, but no differentiation is happening through their internal mathematical operations. This is the subject of ongoing research and will be introduced in future versions of Stride.

From the point of view of the PDE solver, Stride faces the performance-flexibility dichotomy in a similar manner to the geophysical library JUDI [16]: we provide intuitive interfaces in a high-abstraction language, while using a DSL like Devito under the hood. From a symbolic specification of the PDE, Devito automatically generates architecture-specific C code that matches the performance of hand-tuned implementations [17,18]. This offers a high degree of flexibility, allowing the inclusion of new physical models with minimal effort and without hindering performance. It is this flexibility that allows us to run the same wave equation solver on a CPU multi-threaded environment or a GPU with effectively no code changes.

Currently, Stride problems can only be defined on rectangular grids, on which finite-difference methods can be applied using Devito. Nonetheless, Stride does not prescribe any of these, and future work will explore the inclusion of different discretisation approaches and integration with other DSLs like FEniCS/Firedrake for finite-element methods [27,28] or Dedalus for spectral methods [26].

Other open-source libraries exist for numerical modelling in ultrasound medical imaging, such as the previously mentioned kWave [32], based on pseudospectral element methods; Field II [39], which uses a linear scattering approximation; or Bempp-cl [40], which employs a boundary element method, among others. These libraries have been tailored to accurately model sound propagation in biological tissues and generally provide hand-tuned implementations that can achieve high performance. Stride is agnostic to the underlying solver employed and any of these could be readily integrated with it. However, that would diminish the flexibility that is achieved by using a DSL that can obtain comparable performance for both the physical models currently available and any new ones that could be introduced.

Stride has been designed to tackle the problem of intuitively scaling to HPC systems in a similar spirit as for the solver: high-level interfaces hide from the user the complexity of deploying the algorithms to target systems, allowing imaging scientists to focus on the reconstruction algorithms rather than the low-level details. We provide for this the custom parallelisation library Mosaic.

Traditional HPC workloads usually rely on the message passing interface (MPI) standard to express parallelism in applications. However, originally designed in the 1990s, MPI has so far no capacity for fault tolerance and its interfaces are too cumbersome and low level for most non-specialists. Other Python libraries exist for writing parallel applications, most notably Dask [41], PyCOMPSs [42], and Ray [43]. Dask expresses parallelism as a series of stateless tasks that form a computational graph, which can be executed in parallel. PyCOMPSs uses tasks similarly to express parallelism, although these do not have to be stateless. However, PyCOMPSs employs a Java-based runtime that requires the serialisation of objects to file in order to communicate with Python, incurring in a performance penalty. Contrarily, the Ray parallel framework is primarily based on the actor model. We have chosen to design Mosaic using an actor-based model because, much like object-oriented programming, we consider that it better matches the world view and the mental framework of domain specialists. It also allows us to keep objects and their allocated memory warm within a specific compute node or associated accelerator, incidentally making it more intuitive for end users to manipulate remote memory. We have chosen to implement a custom parallelisation library for Stride due to a need for fine-grained control of the computational workload allocation and memory management that existing libraries are unable to provide.

Through the examples presented, we have seen that switching from a local multi-processing environment to an HPC cluster with Mosaic is straightforward and requires no significant code changes. We have also seen through our 3D experiments that realistic Stride reconstructions could be potentially scaled across hundreds of compute nodes thanks to the zero-copy, asynchronous work allocation of the Mosaic library. However, work is still needed to fully understand and exploit the scaling capabilities of Mosaic across large on-premises and cloud computing clusters, with particular interest in minimising data transfers across the network by exploiting caching mechanisms to detect redundant communications.

Additionally, while Mosaic offers the capacity to parallelise across elements of an iteration batch, the integration with Devito offers another degree of freedom to parallelise within PDE solves through MPI-based domain decomposition. Domain decomposition, whose use in Stride is being actively explored, allows a user to distribute the computation of the PDE solution. This will be of importance when solving large problems whose size exceeds memory available in any single node or memory available in a particular accelerator such as a GPU. It will also allow for increased computational performance by splitting PDE solves in a single node across available CPU sockets, thus enforcing data locality.

There are two distinct applications for which Stride has been designed: wave propagation modelling and tomographic imaging. In terms of modelling wave phenomena, a number of other libraries are openly available to users, some of which include the already mentioned Field II [39], Bempp-cl [40], or kWave [32], among others. The choice of one library over another will be down to the aims and requirements of a specific modelling exercise. For example, Field II should be chosen when modelling accuracy can be traded off for shorter computational times, whereas the boundary element method in Bempp-cl will provide accurate modelling that remains computationally efficient when the number of tissue interfaces in the model is low. As we have shown here, Stride and kWave can achieve similar levels of modelling accuracy for complex tissue geometries. Nonetheless, finite-difference solvers in Stride will be more computationally efficient, whereas kWave will display smaller numerical dispersion for a similar discretisation grid thanks to its pseudospectral formulation. These differences will, however, become irrelevant as other numerical methods are integrated into Stride: a different method will be chosen depending on the specific application.

In terms of tomographic imaging, it is important to distinguish between full-wave methods, such as FWI, and others, such as time-of-flight tomography and diffraction tomography. Stride is, at the time of this writing, the only openly available library for full-wave tomographic imaging in the medical context. Stride, however, does not currently provide solvers for other types of ultrasound tomography and other tools should be used in these cases [44].

In terms of compatibility, Stride can be installed on Unix operating systems, and is compatible with Windows through the Windows Subsystem for Linux and through Docker containers.

Through these design decisions, Stride achieves flexibility and modularity, allowing each of its components to be modified independently or entirely substituted. At the same time, importance has been placed on ensuring that lower-level interfaces can be used to provide users with increasingly fine-grained control over the problem and its execution. Although we have designed Stride with ultrasound tomography in mind, the formulation of the physics-constrained optimisation problem is related to other imaging techniques, like optoacoustic tomography, and even calibration methods like spatial response identification. This makes Stride readily applicable to a number of medical ultrasound problems.

5. Conclusions

Advances in ultrasound-based imaging methodologies such as ultrasound computed tomography and optoacoustic tomography

rely on increasingly complex mathematical and computational models. This puts a strain on researchers to both develop novel imaging algorithms and translate them into high-performance and scalable code, thus slowing scientific progress.

To bridge the gap between flexible development and real-life application, we have designed and developed Stride, an open-source Python library that is both intuitive and efficient. Stride allows algorithms to be written for a 2D model and be easily scaled up to 3D, and allows code to be tested on a local workstation and readily deployed to an HPC cluster. We achieve this by combining modular interfaces written in a high-abstraction language with automatically-generated, high-performance solvers, and with tailored parallelisation routines.

By providing high-level interfaces that intuitively match the representation of problems posed by domain specialists, and which are efficient and scalable out of the box, Stride has the potential to dramatically increase the productivity of imaging researchers. This will have a significant impact by accelerating the development of new ultrasound-based imaging technology and its translation from bench to bedside. Furthermore, other imaging applications where the efficient solution of physics-constrained optimisation problems is needed could also benefit from the general abstractions provided by Stride, such as non-destructive testing, aeronautics, or experimental fluid mechanics.

Declaration of Competing Interest

The authors declare no competing interests.

Acknowledgements

This work was supported by the Wellcome Trust [grant number 219624/Z/19/Z]. The work of Carlos Cueto was supported by the Engineering and Physical Sciences Research Council Centre for Doctoral Training in Medical Imaging [grant number EP/L015226/1]. The work of Oscar Bates was supported by the Engineering and Physical Sciences Research Council Centre for Doctoral Training in Neurotechnology [grant number EP/L016737/1]. This work used the ARCHER2 UK National Supercomputing Service. The authors would like to acknowledge the ITRUSST (International Transcranial Ultrasonic Stimulation Safety and Standards) planning group who developed the benchmark example used in Section 3.1 and provided the kWave simulation results. The work of Òscar Calderón Agudo was supported by the UKRI Future Leaders Fellowship [grant number MR/V024086/10].

References

[1] J.W. Wiskin, D.T. Borup, E. Iuanow, J. Klock, M.W. Lenox, 3-D nonlinear acoustic inverse scattering: algorithm and quantitative results, IEEE Trans. Ultrason. Ferroelectr. Freq. Control 64 (8) (2017) 1161–1174, doi:10.1109/TUFFC.2017.2706189.
[2] G.Y. Sandhu, C. Li, O. Roy, S. Schmidt, N. Duric, Frequency domain ultrasound waveform tomography: breast imaging using a ring transducer, Phys. Med. Biol. 60 (14) (2015) 5381–5398, doi:10.1088/0031-9155/60/14/5381.
[3] J. Wiskin, B. Malik, D. Borup, N. Pirshafiey, J. Klock, Full wave 3D inverse scattering transmission ultrasound tomography in the presence of high contrast, Sci. Rep. 10 (1) (2020) 1–14, doi:10.1038/s41598-020-76754-3.
[4] L. Guasch, O. Calderón Agudo, M.-X. Tang, P. Nachev, M. Warner, Full-waveform inversion imaging of the human brain, npj Digit. Med. 3 (1) (2020) 1–12, doi:10.1038/s41746-020-0240-8.
[5] S. Fomel, P. Sava, I. Vlad, Y. Liu, V. Bashkardin, Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments, J. Open Res. Softw. 1 (1) (2013) e8, doi:10.5334/jors.ag.
[6] 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 and Geosciences 85 (2015) 142–154, doi:10.1016/j.cageo.2015.09.015.
[7] R. Hewett, L. Demanet, PySIT: Python seismic imaging toolboxhttps://github.com/pysit/pysit.
[8] D. Koehn, SAVA: 3D seismic modelling, FWI and RTM code for wave propagation in isotropic (visco)-acoustic/elastic and anisotropic orthorhombic/triclinic elastic mediahttps://github.com/daniel-koehn/SAVA.
[9] S. Hassanzadeh, C.C. Mosher, JavaSeis: web delivery of seismic processing services, in: 1997 SEG Annual Meeting, Society of Exploration Geophysicists, 1997, pp. 2055–2057, doi:10.1190/1.1885859.
[10] R.T. Modrak, D. Borisov, M. Lefebvre, J. Tromp, SeisFlows flexible waveform inversion software, Comput. Geosci. 115 (2018) 88–95, doi:10.1016/j.cageo.2018.02.004.
[11] B. Chow, Y. Kaneko, C. Tape, R. Modrak, J. Townend, An automated workflow for adjoint tomography-waveform misfits and synthetic inversions for the North Island, New Zealand, Geophys. J. Int. 223 (3) (2020) 1461–1480, doi:10.1093/gji/ggaa381.
[12] L. Ruthotto, E. Treister, E. Haber, jInv–a flexible Julia package for PDE parameter estimation, SIAM J. Sci. Comput. 39 (5) (2017) S702–S722, doi:10.1137/16m1081063.
[13] C. Da Silva, F. Herrmann, A unified 2D/3D large-scale software environment for nonlinear inverse problems, ACM Trans. Math. Softw. 45 (1) (2019), doi:10.1145/3291042.
[14] L. Krischer, A. Fichtner, S. Zukauskaite, H. Igel, Large-scale seismic inversion framework, Seismol. Res. Lett. 86 (4) (2015) 1198–1207, doi:10.1785/0220140248.
[15] S. Thrastarson, D.-P. van Herwaarden, A. Fichtner, Inversionson: fully automated seismic waveform inversions, EarthArXiv (2021), doi:10.31223/X5F31V.
[16] P.A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G.J. Gorman, F.J. Herrmann, A large-scale framework for symbolic implementations of seismic inversion algorithms in julia, Geophysics 84 (3) (2019) F57–F71, doi:10.1190/geo2018-0174.1.
[17] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P.A. Witte, F.J. Herrmann, P. Velesko, G.J. Gorman, Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration, Geosci. Model Dev. 12 (3) (2019) 1165–1187, doi:10.5194/gmd-12-1165-2019.
[18] F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P.H. Kelly, F.J. Herrmann, G.J. Gorman, Architecture and performance of Devito, a system for automated stencil computation, ACM Trans. Math. Softw. 46 (1) (2020), doi:10.1145/3374916.
[19] S.R. Arridge, M.M. Betcke, B.T. Cox, F. Lucka, B.E. Treeby, On the adjoint operator in photoacoustic tomography, Inverse. Probl. 32 (11) (2016) 115012, doi:10.1088/0266-5611/32/11/115012.
[20] C. Cueto, J. Cudeiro, O.C. Agudo, L. Guasch, M.-X. Tang, Spatial response identification for flexible and accurate ultrasound transducer calibration and its application to brain imaging, IEEE Trans. Ultrason. Ferroelectr. Freq. Control 68 (1) (2021) 143–153, doi:10.1109/TUFFC.2020.3015583.
[21] C. Cueto, L. Guasch, J. Cudeiro, O.C. Agudo, T. Robins, O. Bates, G. Strong, M.-X. Tang, Spatial response identification enables robust experimental ultrasound computed tomography, IEEE Trans. Ultrason. Ferroelectr. Freq. Control 69 (1) (2022) 27–37, doi:10.1109/TUFFC.2021.3104342.
[22] L. Amundsen, Ø. Pedersen, Time step n-tupling for wave equations, Geophysics 82 (6) (2017) T249–T254, doi:10.1190/geo2017-0377.1.
[23] G. Yao, N.V. Da Silva, D. Wu, An effective absorbing layer for the boundary condition in acoustic seismic wave simulation, J. Geophys. Eng. 15 (2) (2018) 495–511, doi:10.1088/1742-2140/aaa4da.
[24] Y. Gao, J. Zhang, Z. Yao, Unsplit complex frequency shifted perfectly matched layer for second-order wave equation using auxiliary differential equations, J. Acoust. Soc. Am. 138 (6) (2015) EL551–EL557, doi:10.1121/1.4938270.
[25] G.J. Hicks, Arbitrary source and receiver positioning in finite-difference schemes using Kaiser windowed sinc functions, Geophysics 67 (1) (2002) 156–166, doi:10.1190/1.1451454.
[26] K.J. Burns, G.M. Vasil, J.S. Oishi, D. Lecoanet, B.P. Brown, Dedalus: a flexible framework for numerical simulations with spectral methods, Phys. Rev. Res. 2 (2) (2020) 23068, doi:10.1103/physrevresearch.2.023068.
[27] A. Logg, K.A. Mardal, G. Wells, Automated solution of differential equations by the finite element method: the FEnics book, Lecture Notes in Computational Science and Engineering, vol. 84, 2012, doi:10.1007/978-3-642-23099-8.
[28] F. Rathgeber, D.A. Ham, L. Mitchell, M. Lange, F. Luporini, A.T. McRae, G.T. Bercea, G.R. Markall, P.H. Kelly, Firedrake: automating the finite element method by composing abstractions, ACM Trans. Math. Softw. 43 (3) (2016) 24, doi:10.1145/2998441.
[29] ZeroMQ Development Team, ZeroMQ: an open-source universal messaging libraryhttps://zeromq.org/.
[30] The HDF Group, Hierarchical Data Format Version 5, Technical Report, http://www.hdfgroup.org/HDF5.
[31] P. Morse, H. Feshbach, Methods of theoretical physics, no. v. 1 in International Series in Pure and Applied Physics, McGraw-Hill, 1953. https://books.google.co.uk/books?id=atwEAAAACAAJ
[32] B.E. Treeby, B.T. Cox, k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields, J. Biomed. Opt. 15 (2) (2010) 021314, doi:10.1117/1.3360308.
[33] H. Vogel, A better way to construct the sunflower head, Math. Biosci. 44 (3–4) (1979) 179–189, doi:10.1016/0025-5564(79)90080-4.
[34] J.-F. Aubry, O. Bates, C. Boehm, K.B. Pauly, D. Christensen, C. Cueto, P. Gelat, L. Guasch, J. Jaros, Y. Jing, R. Jones, N. Li, P. Marty, H. Montanaro, E. Neufeld, S. Pichardo, G. Pinton, A. Pulkkinen, A. Stanziola, A. Thielscher, B. Treeby, E.v.t. Wout, Benchmark problems for transcranial ultrasound simulation: intercomparison of compressional wave models, ArXiv (2022), doi:10.48550/arxiv.2202.04552.
[35] Y. Lou, W. Zhou, T.P. Matthews, C.M. Appleton, M.A. Anastasio, Generation of anatomically realistic numerical phantoms for photoacoustic and ultrasonic breast imaging, J. Biomed. Opt. 22 (4) (2017) 041015, doi:10.1117/1.JBO.22.4.041015.

[36] A.J. Chee, C.K. Ho, B.Y. Yiu, A.C. Yu, Walled carotid bifurcation phantoms for imaging investigations of vessel wall motion and blood flow dynamics, IEEE Trans. Ultrason. Ferroelectr. Freq. Control 63 (11) (2016) 1852–1864, doi:10.1109/TUFFC.2016.2591946.

[37] M.I. Iacono, E. Neufeld, E. Akinnagbe, K. Bower, J. Wolf, I. Vogiatzis Oikonomidis, D. Sharma, B. Lloyd, B.J. Wilm, M. Wyss, K.P. Pruessmann, A. Jakab, N. Makris, E.D. Cohen, N. Kuster, W. Kainz, L.M. Angelone, MIDA: a multimodal imaging-based detailed anatomical model of the human head and neck, PLoS ONE 10 (4) (2015) e0124126, doi:10.1371/journal.pone.0124126.

[38] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Automatic differentiation in PyTorch, in: Conference on Neural Information Processing Systems (NIPS), 2017.

[39] J.A. Jensen, FIELD: a program for simulating ultrasound systems, in: 10th Nordic Baltic Conference on Biomedical Imaging, vol. 34, 1996, p. 353.

[40] T. Betcke, M. Scroggs, Bempp-cl: a fast Python based just-in-time compiling boundary element library, J. Open Source Softw. 6 (59) (2021) 2879, doi:10.21105/joss.02879.

[41] Dask Development Team, Dask: library for dynamic task scheduling, 2016, https://dask.org.

[42] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R.M. Badia, J. Torres, T. Cortes, J. Labarta, PyCOMPSs: parallel computational workflows in Python, Int. J. High Perform. Comput. Appl. 31 (1) (2017) 66–82, doi:10.1177/1094342015594678.

[43] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: a distributed framework for emerging AI applications, in: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, 2017, pp. 561–577.

[44] R. Ali, S. Hsieh, J. Dahl, Open-source Gauss-Newton-based methods for refraction-corrected ultrasound computed tomography, in: SPIE Medical Imaging, vol. 10955, SPIE, 2019, pp. 39–52, doi:10.1117/12.2511319.