
Φ_{Flow} : Differentiable Simulations for PyTorch, TensorFlow and Jax

Philipp Holl¹ Nils Thuerey¹

Abstract

Differentiable processes have proven an invaluable tool for machine learning (ML) in scientific and engineering settings, but most ML libraries are not primarily designed for such applications. We present Φ_{Flow} , a Python toolkit that seamlessly integrates with PyTorch, TensorFlow, Jax and NumPy, simplifying the process of writing differentiable simulation code at every step. Φ_{Flow} provides many essential features that go beyond the capabilities of the base libraries, such as differential operators, boundary conditions, the ability to write dimensionality-agnostic code, floating-point precision management, fully differentiable preconditioned (sparse) linear solves, automatic matrix generation via function tracing, integration of SciPy optimizers, simulation vectorization, and visualization tools. At the same time, Φ_{Flow} inherits all important traits of the base ML libraries, such as GPU / TPU support, just-in-time compilation, and automatic differentiation. Put together, these features drastically simplify scientific code like PDE or ODE solvers on grids or unstructured meshes, and Φ_{Flow} even includes out-of-the-box support for fluid simulations. Φ_{Flow} has been used in various publications and as a ground-truth solver in multiple scientific data sets. It is available at <https://github.com/tum-pbs/PhiFlow>.

1. Introduction

The combination of deep learning and physics simulations has sparked a multitude of promising lines of research. Neural networks have been used to accelerate simulations (Sanchez-Gonzalez et al., 2020; Tompson et al., 2017), improve simulation accuracy for fixed resolutions (Kochkov

et al., 2021b; Um et al., 2020), control complex physical systems (Bieker et al., 2020; Holl et al., 2020), encode physical states and sequences (Raissi et al., 2019) and find conservation laws (Greydanus et al., 2019), among others. In all of these tasks, the output of a neural network is interpreted as a physical quantity that influences a physical system and the learning objective is defined in terms of the resulting physical behavior, i.e. the network prediction passes through a physics simulation before the loss is defined. Optimal training methods for the first three tasks even require the prediction to go through multiple simulation steps, passing through the network multiple times along the way (Um et al., 2020).

Training a neural network to directly minimize the loss for any of the above-mentioned tasks requires the physical behavior to be differentiable in order to compute the derivative of the loss w.r.t. the network weights. Differentiable simulations have long been used in classical optimization (Jarny et al., 1991; Jameson, 2003) where the adjoint method is typically employed to compute the required gradients (Plessix, 2006). The adjoint method is also used in machine learning (ML), where it is known as reverse-mode differentiation or simply backpropagation. Consequently, it can also be used to backpropagate through joint systems comprising of both neural networks and physical simulations as long as all parts are differentiable.

Despite this deep connection between the two fields, most established software frameworks focus on only one of them. There are frameworks for differentiable simulations (Todorov et al., 2012; Mitusch et al., 2019) and separate frameworks for neural network optimization, such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016), and Jax (Bradbury et al., 2018b; Babuschkin et al., 2020; Hennigan et al., 2020). Combining these frameworks is hard to achieve in practice and many researchers have instead chosen to implement custom differentiable simulations compatible with one specific machine learning framework (Tompson et al., 2017; Kochkov et al., 2021b; Bieker et al., 2020). However, this approach results in highly specialized and low-level simulation code, preventing adoption to different projects.

A number of libraries combine differentiable physics and ML (Schoenholz & Cubuk, 2020; Hu et al., 2019; Bezin

*Equal contribution ¹School of Computation, Information and Technology, Technical University of Munich, Germany. Correspondence to: Philipp Holl <philipp.holl@tum.de>.

et al., 2022; Kochkov et al., 2021a; Macklin, 2022) but they are either very specialized or use different programming paradigms than the popular ML libraries, making seamless integration difficult. The programming language Julia (Bezanson et al., 2017) offers language-level differentiation but is not compatible with most established ML libraries. The difficulty in training neural networks with differentiable physics has led many authors to fall back to supervised learning (Sanchez-Gonzalez et al., 2020; Rasp & Thuerey, 2021; Thapa et al., 2020; Stachenfeld et al., 2021).

In this work, we present Φ_{Flow} (PhiFlow), an open-source framework for differentiable simulations that builds on top of PyTorch, TensorFlow, Jax or NumPy (Harris et al., 2020), using the abstraction layer Φ_{ML} (Holl & Thuerey, 2024). It is intended to be used for a wide variety of simulations and includes high-level data structures for grid-based (Eulerian) as well as particle-based (Lagrangian) simulations. Φ_{Flow} is designed to make simulation code as reusable as possible without sacrificing readability or performance. Additionally Φ_{Flow} aims to accelerate development iterations by promoting interactivity and clean code. Originally developed as a fluid solver in 2019, Φ_{Flow} has since accumulated many features and extended its scope. It has been used in production for multiple works and publications (Um et al., 2020; Holl et al., 2020; 2021; Thuerey et al., 2021; Schnell et al., 2022; Holl et al., 2022; Brandstetter et al., 2021; Wandel et al., 2020b; Brandstetter et al., 2022; Takamoto et al., 2023; Teikmanis et al.; Wandel et al., 2021; 2020a; Bokil et al., 2023; Sengar et al., 2021; Parekh et al., 1993; Brahmachary & Thuerey, 2023; Ramos et al., 2022; Yadav et al., 2021; Tathawadekar et al., 2021; Wang et al., 2022a; Kurz et al., 2023; Wang et al., 2022b; Wang, 2023; Wu et al., 2022; Li et al., 2023), as well as open data sets (Takamoto et al., 2022; Gupta & Brandstetter, 2022). This paper outlines the following contributions:

- Φ_{Flow} includes a fully-featured Eulerian incompressible fluid solver. All operations, e.g. diffusion and convection, can be repurposed for other simulations, such as heat transfer.
- Φ_{Flow} provides differential operators on grids and unstructured meshes, including higher-order spatial differentiation on grids via compact schemes.
- Φ_{Flow} includes boundary condition handling for all common types of boundaries, allowing boundary types to be changed without altering the simulation code.
- We introduce a system for writing dimensionality-agnostic code by tagging dimensions as spatial axes, so user code can run in 1D, 2D, 3D, and higher dimensions without modification.
- We introduce a new system for handling floating-point precision to simplify specifying the desired accuracy and avoid data-type-related errors.
- Φ_{Flow} includes differentiable linear solvers and preconditioners for dense and sparse matrices. Unlike other libraries, we can also differentiate w.r.t. the matrix and all of its dependencies.
- We introduce automatic matrix generation via function tracing. Φ_{Flow} can automatically build matrices from linear Python functions by recording the performed operations.
- Φ_{Flow} seamlessly integrates all linear and non-linear SciPy optimizers into all supported ML libraries in an efficient manner.
- We introduce an alternative way of vectorizing code by defining batch dimensions instead of using `vmap`. This preserves break points for debugging and is more flexible than `vmap`.
- We introduce a plotting library that can create a broad range of plots with a single call, drastically reducing the overhead required for visualizing results and intermediate quantities.

We first explain the major features included in Φ_{Flow} before demonstrating their application on challenging problems involving simulations and neural networks. We show that using Φ_{Flow} leads to more flexible, concise, readable, and less error prone code compared to directly using the base libraries, without compromising performance.

2. Major features of Φ_{Flow}

All of Φ_{Flow} 's core functionality is implemented directly in Python 3 (Van Rossum & Drake Jr, 1995). This makes it compatible with PyTorch, TensorFlow and Jax, and allows users to easily locate and understand the implementation of all provided functions. To make this broad range of compatibility possible, Φ_{Flow} employs an internal wrapping layer that abstracts the tensor operations provided by the chosen ML library or NumPy. All functions in Φ_{Flow} build upon this abstraction and are thus compatible with all backends. DLPack (et al., 2017) integration further allows users to switch between ML libraries mid-execution and without copying data.

Φ_{Flow} provides classes to represent various geometries, such as grids, unstructured meshes, or point clouds, and a class to represent physical quantities, *fields*, sampled on these geometries. In the following, we go over the major features that Φ_{Flow} adds on top of the base ML libraries, along with example code. In fact, we provide the full source code of all main experiments in this document to demonstrate the brevity of code written against our API.

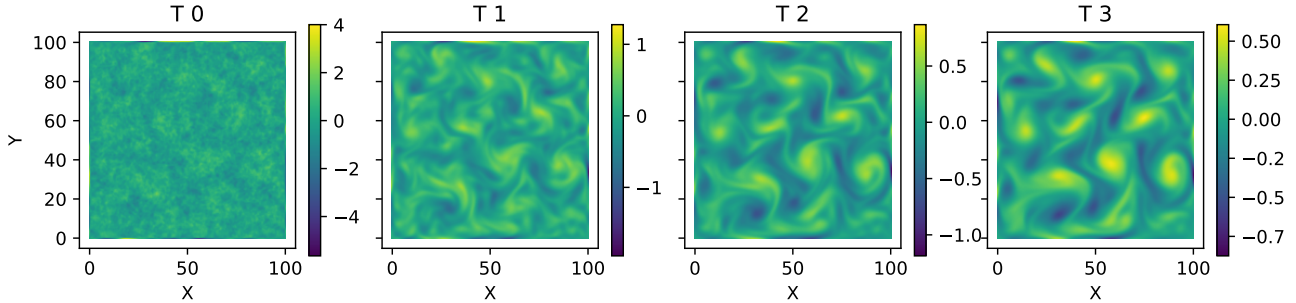


Figure 1. Snapshots of the vorticity field $\nabla \times \vec{v}$ of the incompressible fluid simulation defined in Fig. 2. The full code generating this figure is `plot(v.trj.t[1::25].curl(), same_scale=False, size=(10, 2.5))`. All labels and bars are added automatically.

Fluid solver Incompressible fluid simulations are challenging, both theoretically and numerically (Ferziger et al., 2019). Unlike many other toolkits, Φ_{Flow} does not provide a stand-alone solver but rather a set of building blocks from which a full simulation can quickly be assembled. This gives users full control and makes it easy to adapt the code to their specific needs. The example code in Fig. 2 runs an incompressible fluid simulation using staggered grids and operator splitting. To extend this to a smoke simulation with buoyancy, for example, we can extend this function to also advect a smoke density field s and add the resulting buoyancy to the velocity field via $v += dt * \text{resample}(s * (0, 0.1), t_0=v)$. A visualization of this simulation is shown in Fig. 1.

```

1 from phi.torch.flow import *
2 @jit_compile
3 def incompressible_fluid_step(v, dt=.25):
4     v, p = fluid.make_incompressible(v)
5     v = diffuse.explicit(v, 0.1, dt)
6     v = advect.semi_lagrangian(v, v, dt)
7     return v
8 v0 = StaggeredGrid(Noise(), x=256, y=256, bounds=Box(x=100, y=100), boundary=0)
9 v_trj = iterate(incompressible_fluid_step, batch(t=100), v0)

```

Figure 2. Executable source code of an incompressible (line 4) fluid simulation consisting of diffusion (line 5) and advection (line 6) with operator splitting. The simulation (lines 3-7) runs completely on PyTorch (line 1) with just-in-time compilation (line 2) and will make use of GPU acceleration if available. We initialize an initial velocity from random noise fluctuations on a staggered grid with 256×256 cells spanning 100 physical units (line 8) and record the simulation for 100 time steps (line 9) with $\Delta t = 0.25$. The quantity p (line 4) denotes the pseudo-pressure from the Helmholtz decomposition of the velocity v . Fig. 1 visualizes this simulation.

Differential operators Differential operators are a key building block of all PDEs, and many such simulations can be expressed as purely a combination of applying differential operators and basic arithmetic. Φ_{Flow} implements all common differential operators for grids as well as unstructured meshes. These includes the gradient ∇u , divergence

$\nabla \cdot \vec{u}$, laplace $\nabla^2 u$, and curl $\nabla \times \vec{u}$, as well as their generalizations to matrix fields. For grids, Φ_{Flow} also provides higher-order operators, letting users specify the spatial order of accuracy via the `order` keyword, e.g. `divergence(v, order=6)`. We generate the corresponding stencils on-the-fly, allowing users to specify any even order of accuracy. All differential operators are compatible with all supported boundary conditions, which are automatically inferred from the field.

Boundary conditions Correctly handling boundary conditions can be a major hurdle in writing simulations, as the implementation depends not only on the type of boundary but also the employed numerical scheme and the nearby topology. Higher-order methods may necessitate the use of one-sided stencils close to the boundary while lower-order methods can employ ghost cells. Furthermore, boundary conditions affect particle operations differently than grid operations, e.g. how to compute distances or which points are relevant for interpolation. Φ_{Flow} alleviates these complications by providing an extensive library dedicated to boundary condition handling. Along with the supported types (Dirichlet, Neumann, periodic, symmetric conditions), Φ_{Flow} includes boundary condition arithmetic, padding and distance functions, as well as index transforms. Users need only specify the boundary for each field, optionally per-side, and all physics function will automatically adjust the employed numerical scheme accordingly. E.g., to specify that field values should be zero at the boundary, pass `boundary=0` as in Fig. 2 line 8, or specify `boundary='periodic'` for periodic boundary conditions.

Dimensionality-agnostic code Many PDEs can be realized in multiple dimensionalities, i.e. 1D, 2D, 3D, etc., because most differential operators are dimension-agnostic. Take the heat equation, $\dot{u} = \kappa \nabla^2 u$ for example, where the Laplace operator $\nabla^2 = \sum_i \frac{\partial^2 u}{\partial x_i^2}$ can be computed in any number of dimensions. This abstract mathematical description usually does not translate into code, however, as tensor shapes and ranks vary with the dimensionality, requiring

specific handling for each realization. We introduce a new system for writing dimensionality-agnostic code. Instead of specifying the axes to operate on per operation, users declare tensor dimensions as *spatial* dimensions. All relevant functions, such as differential operators or spectral operations, can then infer the dimensionality from the number of tagged dimensions. This allows users to test their 3D simulations in 2D without changing the simulation function. The code given in Fig. 2 can be made to run in 1D, 2D, 3D, and higher dimensions, by modifying only line 8. No change to the simulation function is required.

Floating-point precision management All common numerics and ML libraries infer the precision of tensor operations from the inputs, upgrading values or raising errors when data types do not match exactly. This approach comes with two major problems: (i) porting code to use a different floating-point (FP) precision requires changing the data types of all involved tensors, which can incur changes in many places, and (ii) automatic upgrading can lead to unintentional type conversions, making it hard to track the root cause. This issue has led the Jax developers to completely disable double precision by default. Since controlling the FP precision is essential for numerical simulations, we have developed a new system to solve these issues. Φ_{Flow} 's tensor operations determine the desired FP precision from the operation context rather than the data types of its inputs. The precision can be set globally or specified locally via context managers, and operations will automatically convert tensors of non-matching data types if necessary. This avoids data-type-related problems and errors, as well as making user code more concise and cohesive. To make the example in Fig. 2 run with double precision, insert the line `math.set_global_precision(64)` below line 1.

Fully differentiable preconditioned linear solvers Solving systems of linear equations $Ax = b$ is a vital function for numerical simulations, such as for computing implicit operations. Unlike explicit computations, they guarantee numerical stability, making them a popular choice in many methods, such as FEM (Reddy, 2019), FVM (Versteeg & Malalasekera, 2007) or MPM (Bardenhagen et al., 2004). Φ_{Flow} includes all SciPy solvers, as well as custom GPU-compatible conjugate gradient and (stabilized) bi-conjugate gradient methods (Shewchuk et al., 1994) for solving linear systems of equations, both with sparse and dense matrices. Furthermore, Φ_{Flow} comes with support for GPU-compatible preconditioners, such as the incomplete LU decomposition (Saad, 1996) and clustering, which can drastically improve convergence speed. Linear solves are implemented differentially, i.e. the adjoint system is solved during back-propagation or computation of higher-order derivatives. In addition to differentiating w.r.t. the right-hand-side b , Φ_{Flow} can also differentiate w.r.t. the (sparse) matrix A and all of

its dependencies, a feature that is missing from the base ML libraries but is required in many circumstances, e.g. when differentiating through implicit convection or finding the optimal viscosity in implicit diffusion.

Automatic matrix generation via function tracing

There are generally two categories of linear system solvers: ones that use an explicit representation of the matrix A , and matrix-free solvers which use a functional representation $\hat{A}(x)$ that computes the result of $A \cdot x$. While the latter is more convenient, concise, readable and debuggable in code, the former is more efficient and enables usage of generic preconditioners, such as the incomplete LU decomposition of A . We aim to combine the best of both worlds by introducing automatic (sparse) matrix generation. This allows users to write and test the effect of A as a function but still perform an explicit matrix solve with automatically-generated preconditioners. To achieve this, we implement a function tracing algorithm similar to just-in-time compilation, but, instead of low-level code, it outputs a matrix, representing the effect of the function \hat{A} on a placeholder vector. It records all affine operations, including boundary conditions, performed by \hat{A} and assembles the matrix $A \in \mathbb{R}^{n \times m}$ and offset $o \in \mathbb{R}^n$, such that $\hat{A}(x) = A \cdot x + o \forall x \in \mathbb{R}^m$. Tracing can be done explicitly by the user via `matrix_from_function()`, or under-the-hood by decorating an affine Python function with `@jit_compile_linear`. Then the matrix will be generated when the function is used in a linear solve, and all constant terms in \hat{A} will be automatically subtracted from the right-hand-side vector to solve $A \cdot x = b - o$. Tracing Python functions may seem like a large overhead if it needs to be done for each simulation step, but, in addition to caching, Φ_{Flow} includes various optimizations for production code, i.e. when the simulation is jit-compiled.

- The dependencies of A and o on simulation parameters are expressed as a jit-compiled computational graph in the corresponding ML library, reducing the overhead of matrix construction.
- The sparsity pattern of A is determined at jit-compile time, usually performed only once. For variable patterns, such as upwind schemes, the combined pattern is determined and zeros are added to the matrix values where necessary.
- Sparse matrices are automatically compressed at compile-time into the most optimal format supported by the ML library, such as the compressed sparse row (CSR) format. At runtime, this only induces a gather operation with fixed indices on the values tensor to order the entries correctly.
- If \hat{A} has no dependence on variables outside x , the

matrix is computed using NumPy. It enters the computational graph of the ML library as a constant, and no matrix-building operations need to be performed at runtime.

- If the matrix values depend on parameters that vary across examples, the sparse matrix can be represented in either monolithic block-diagonal form or batched-values form to maximize hardware utilization.

These optimizations also apply to all preconditioners.

Integration of SciPy optimizers Scientific problems often require the use of general function minimizers. Φ_{Flow} includes a simple API for the SciPy optimizers available through `scipy.optimize.minimize`, like the popular L-BFGS method. For example, `minimize(sin, Solve('BFGS', x0=0))` returns the minimum of $\sin(x)$ using BFGS with initial guess 0. The full optimization trajectory can optionally be recovered as well by using a `SolveTape`. Φ_{Flow} integrates the SciPy methods into the computational graph of the base ML library, i.e. derivatives are computed via automatic differentiation and can be evaluated on the GPU. Φ_{Flow} also supports solving batches of optimization problems in parallel by bundling the current estimates of the individual optimizations.

Vectorization via batch dimensions PyTorch, TensorFlow, and Jax all support two ways of vectorizing code: batch dimensions and function transformations, i.e. `vmap`. Both often exhibit equal performance, but operations supporting batch dimensions are always implemented in the most efficient manner, which is not guaranteed for `vmap`. When it comes to code development and maintenance, batch dimensions also have a major advantage over `vmap` as they support break points for debugging, visualizing the batched quantities, and easily computing batch statistics. However, the base ML libraries only support batch dimensions for a subset of their API. This is especially relevant for simulation code which frequently makes use of more niche operations. Φ_{Flow} 's API supports batch dimensions for *all* operations, making `vmap` superfluous. Our system is implemented so that all user-written code can be trivially vectorized by adding one or more batch dimensions to any of its inputs. We achieve this by letting users tag tensor dimensions as batch dimensions, a property that is retained throughout the whole computational graph. For example, to simulate a batch of four fluid simulations with different initial conditions, simply pass `batch(b=4)` as an argument to `Noise()` in line 8 of Fig. 2. This adds a batch dimension with name `b` to the initial velocity field and propagates it through the simulation code.

Single-call visualization A host of scientific plotting libraries can be used to create high-quality figures and anima-

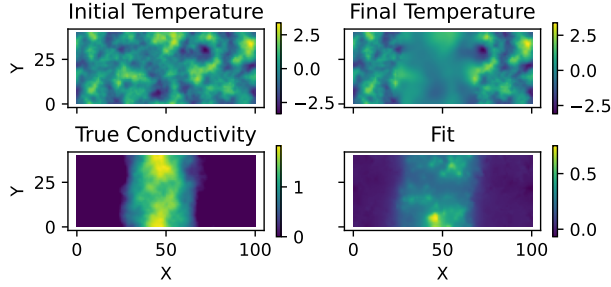
tions. Popular libraries, such as Matplotlib (Hunter, 2007) and Plotly (Inc., 2015) provide a multitude of functions to configure every aspect of the plot. This flexibility comes at the cost of verbose user code, however. A Matplotlib plot with titled subfigures and labelled axes typically requires a large paragraph of code to create. This level of effort is acceptable for final plots, but makes intermediate visualizations tedious. To alleviate this issue, we have developed a plotting frontend for Matplotlib and Plotly that creates fully-fledged figures with a single command: `plot()`. This is possible due to the support for dimension names and types that all Φ_{Flow} tensors and composite types have. For example, the batch dimensions introduced above translate to subfigures by default but can alternatively be used as the time axis in an animation. Based on the types and names of the data dimensions, Φ_{Flow} selects an appropriate plot type to represent the data, automatically labelling all axes according to the names given to the tensor dimensions. Currently available plot types for tensors include line plots, bar charts, histograms, 2D and 3D heatmaps, 2D and 3D vector fields, stream plots and scatter plots. All supported geometry types can also be shown, additionally enabling grid and mesh visualizations, 1D and 2D heightmap plots, as well as geometric primitives, like circles, rectangles, cuboids, and spheres. The `plot()` function offers some options to customize plots, such as adding error bars or specifying colors, but Φ_{Flow} 's plotting API only consists of `plot()` and `show()`. This makes it extremely easy to use and, since the created figures are Matplotlib or Plotly types, those libraries can be used to make alterations if required. Fig. 1 shows snapshots of the fluid simulation from above. The whole figure was generated with one line of code. All other plots shown in this document were also created with one `plot()` call.

3. Experiments

To illustrate how the above features can be used, not only to simulate ground-truth data, but also to solve complex inverse problems, we perform a series of challenging experiments and reimplement experiments from prior work. We provide the full source code including data generation in the corresponding figures and all shown plots are generated with our Matplotlib frontend. Jupyter notebooks containing the source code together with all plotting code are available in the supplemental information (SI), and performance measurements are given in appendix A.

3.1. Material composition from thermal conductivity

First, we consider a heat conduction setting. A plane consists of a mixture of two materials with different thermal conductivity coefficients, e.g. a conductor and an insulator. The task is to determine the fraction of each material at



```

1 from phi.jax.flow import *
2 def simulate(x): # x is the guess for the conductivity
3     return diffuse.implicit(initial_temp, field.maximum(0, x), 10, Solve('biCG-stab(2)'))
4 boundary = {'x': 'periodic', 'y': 'zero-gradient'}
5 conductivity = field.maximum(0, CenteredGrid(Noise(scale=100), boundary, x=100, y=40))
6 initial_temp = CenteredGrid(Noise(), boundary, x=100, y=40)
7 final_temp = simulate(conductivity)
8 fit = minimize(lambda x: math.l2_loss(final_temp - simulate(x)),
                Solve('GD', x0=conductivity.with_values(.01)))
    
```

Figure 3. Visualization and full source code of the conductivity reconstruction experiment, including data generation (lines 5-7) and optimization (line 8). We consider a 2D domain with 100×40 cells (line 5-6) with periodic boundary conditions on the left and right edge and open top and bottom boundaries (line 4). The ground truth conductivity is initialized from large-scale fluctuations clipped to positive values (line 5), and the observed initial temperature distribution is sampled from smaller-scale noise (line 6). Using the simulation function (line 2), the observed temperature ten seconds later (line 7) is computed via an implicit diffusion operation employing a GPU-enabled biconjugate gradient solver (line 3). We reconstruct the conductivity using steepest gradient descent (line 8), minimizing the mean squared error between observed final state and simulation output. The optimizer uses the automatic differentiation provided by the Jax library (line 1).

every point. To achieve this, we warm up some places and observe the so-created temperature profile, as well as the profile at a later time. Reconstructing the exact material fractions from these two snapshots alone is impossible as only variations in the initial temperature gradient yield information about the composition. Furthermore, as the temperature approaches equilibrium, most information about the composition is erased. Despite these fundamental challenges, we aim to obtain a best guess of the material component fractions.

We implement this scenario by generating a ground truth conductivity C and initial temperature profile T_0 (Fig. 3 left plots). Using Φ_{Flow} , we can easily write a differentiable simulator for implicit heat conduction, ensuring numerical stability for all simulated times (Fig. 3 code). We use this to generate the ground-truth observation data T_f and to optimize for the material composition. Using the mean squared error (MSE) between observed and reconstructed temperature profile $T(T_0, \Delta t, C)$, we need to solve the following

inverse problem.

$$C = \arg \min_{C'} \|T(T_0, \Delta t, C') - T_f\|_2^2 \quad (1)$$

We employ steepest gradient descent to approximate C . This requires the gradient $\frac{\partial T(\cdot, C)}{\partial C}$, i.e. differentiating the implicit diffusion result T by the composition C which only affects the matrix entries, not the right-hand-side vector. This would require manual gradient implementation in most ML libraries, but Φ_{Flow} includes support for this kind of differentiation.

Fig. 3 shows an example reconstruction. As expected, the fine detail cannot be recovered but the fit adequately approximates the ground truth conductivity distribution. This whole experiment can be expressed in eight lines of Python code when using Φ_{Flow} , attesting to the high information-to-code ratio. We encourage readers to go through the source code line-by-line, as we believe it explains our methodology in more detail and clarity than we can achieve in text.

3.2. Particle image velocimetry

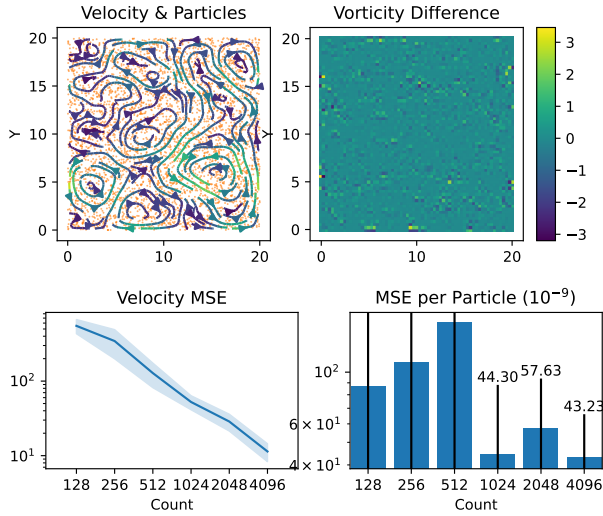
Reconstruction the motion of a fluid can be done by tracking the positions of small marker particles, i.e. particle image velocimetry (PIV). The markers are passively advected with the fluid, and, given the particle positions at two consecutive frames, one can fit the velocity field at that time. We realize a PIV solver using Φ_{Flow} 's differentiable advection operation. With the objective to minimize the observed marker positions $M(M_0, \Delta t, v)$ on the second frame given the initial positions M_0 , we get the inverse problem

$$v = \arg \min_{v'} \|M(M_0, \Delta t, v') - M_0\|_2^2, \quad (2)$$

where v denotes the fluid velocity. We first fit a coarse velocity grid at quarter-resolution to avoid zero-velocity values in cells empty of markers, and then perform a residual fit at full resolution. For both fits, we employ SciPy's L-BFGS-B optimizer, which converges significantly faster than gradient descent. The top plots of Fig. 4 show an example velocity field with 4096 markers and the reconstruction error.

To determine how many markers are required to adequately reconstruct v with this algorithm, we perform this experiment for multiple numbers of marker particles and test 16 different velocity fields, each. Using Φ_{Flow} 's batch dimensions, we can simply expand the relevant simulation inputs and run the optimization and simulation code without modification, as can be seen in Fig. 4, where the two batch dimensions `seed` and `count` are introduced in lines 5 and 7. Varying the number of particles does change the tensor sizes, but all sizes are still tracked consistently throughout the simulation. The bottom plots in Fig. 4 show that increasing the number of markers improves the velocity MSE at about one order of magnitude per $8 \times$ more markers. The

average particle position MSE also decreases but is more variable across runs with large standard deviations.



```

1 from phi.jax.flow import *
2 @jit_compile
3 def simulate(v): # v is the guess for the velocity
4     return advect.points(initial_markers, v, dt=1, integrator=advect.rk4)
5 v0 = StaggeredGrid(Noise(batch(seed=16)), x=64, y=64, bounds=Box(x=20, y=20))
6 v0._ = fluid.make_incompressible(v0)
7 marker_count = vec(batch('count'), 128, 256, 512, 1024, 2048, 4096)
8 initial_markers = v0.bounds.sample_uniform(instance(markers=marker_count))
9 final_markers = simulate(v0)
10 fit1 = minimize(lambda x: math.l2_loss(final_markers - simulate(x)),
11                Solve('L-BFGS-B', x0=0 * v0.downsample(4)), at(v0))
12 fit2 = minimize(lambda x: math.l2_loss(final_markers - simulate(x+fit1)),
13                Solve('L-BFGS-B', x0=0 * v0))
14 v_estimate = fit1 + fit2
    
```

Figure 4. Visualization and full source code of the PIV experiment, including data generation (lines 5-9) and optimization (lines 10-11). We generate a divergence-free 64×64 ground-truth velocity field v_0 (lines 5-6, top-left plot) and randomly distribute a number of particles within it (lines 7-8, orange points). The observed particle positions 0.1 seconds later (line 9) are determined using a Runge-Kutta-4 (Runge, 1895) advection scheme (line 4). To reconstruct the velocity field from the observed particle movement, we perform two L-BFGS-B optimizations, one at quarter-resolution (line 10) and a residual fit at full resolution (line 11). The reconstructed velocity is the sum of the upscaled coarse and fine residual fits (line 12). The vorticity difference between ground truth velocity and reconstruction is shown in the top right. We perform this experiment for various numbers of particles between 128 and 4096 (line 7) and 16 different velocity initialization seeds (line 5) in parallel. The lower plots show the reconstruction errors in velocity and particle spaces depending on the particle count (mean and standard deviation).

3.3. Learning to simulate fluids

Training neural networks to mimic ground-truth simulators is a major goal in scientific machine learning (), as they may achieve similar results in a fraction of time. Multiple schemes for training the networks have been investigated,

and it has become increasingly clear that a one-step supervised objective is not ideal (Um et al., 2020). Rolling out multiple time steps at training time can drastically increase long-term stability and prevent numerical explosions, as the network learns to account for the errors made by its previous iterations (data shift).

Here, we train a U-Net (Ronneberger et al., 2015) N to mimic the fluid solver defined in Fig. 2. Given the initial state u_0 and consecutive frame $u_1 = S(u_0)$ computed using the ground-truth simulator, we define the loss function for the network N_θ parameters

$$L(\theta) = \|N_\theta(u_0) - u_1\|_2^2 + \lambda \|N_\theta^2(u_0) - S(N_\theta(u_0))\|_2^2, \quad (3)$$

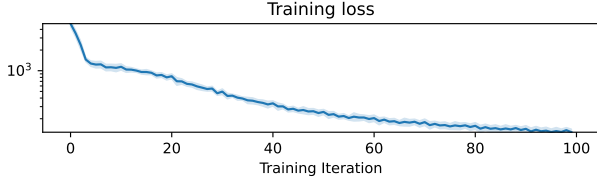
where N^2 denotes applying the network twice. The first term represents a one-step supervised loss and the second term computes the error after two time steps. Accurate gradients $\frac{\partial L}{\partial \theta}$ require a differentiable simulator S for the second term. The implementation of this training setup, along with example learning curves, are shown in Fig. 5. We use Φ_{Flow} 's built-in U-Net implementation which is available for PyTorch, Jax, and TensorFlow. Like the similar convolutional, residual, and fully-connected architectures, this U-Net can be created with one line of code and is configurable in numbers of inputs and outputs, intermediate layers, activations functions, and more.

After an initial drop, the learning curve in Fig. 5 exhibits linear convergence, i.e. the loss decreases exponentially with the number of iterations. This is expected for this optimization using Adam (Kingma & Ba, 2014) and shows that the network can be successfully trained with the differentiable simulator.

3.4. Billiards, sphere packing & wave propagation

In addition to our main experiments, we replicate three setups from prior work to show that Φ_{Flow} 's features scale to problems outside its core domain. Specifically, we implement (i) the billiards experiment from (Hu et al., 2019) which simulates collisions between balls, (ii) the sphere packing experiment from (Schoenholz & Cubuk, 2020) which searches for the optimal configuration of spheres with minimal overlap, and (iii) the wave propagation example from (Macklin, 2022) which performs a 2D grid simulation. All three problems were published as showcases for their respective frameworks. To assess the readability of our implementation, we tasked ChatGPT with explaining our code and the original code, see Appendix B.

Billiards This experiment served as a demonstration of differentiable collisions. Ten billiard balls are placed in a triangular formation, and a cue ball is placed some distance apart, as shown in Fig. 6. The task is finding the initial velocity of the cue ball v_{cue} , such that one corner ball from



```

1 from phi.jax.stax.flow import *
2 net = u_net(in_channels=2, out_channels=2, levels=4)
3 opt = adam(net, learning_rate=1e-2)
4 def loss_function(x: Field, y1):
5     pred1 = field.native_call(net, x)
6     y2 = incompressible_fluid_step(pred1)
7     pred2 = field.native_call(net, pred1)
8     return math.l2_loss(pred1 - y1) + .5 * math.l2_loss(pred2 - y2)
9 x = StaggeredGrid(Noise(batch(dataset=10, batch=10)), x=63, y=63)
10 y = incompressible_fluid_step(x)
11 for epoch in range(10):
12     for xb, yb in zip(x.dataset, y.dataset):
13         print(update_weights(net, opt, loss_function, xb, yb))
    
```

Figure 5. Visualization and source code for training a U-Net (line 2) using Adam (line 3) to mimic an incompressible fluid simulation. Based on the first prediction $t_0 \rightarrow t_1$, we run the neural network for a second time $t_1 \rightarrow t_2$ and simultaneously perform a time step with the ground-truth simulator from Fig. 1 to improve long-term stability (lines 6-7). The objective (line 4) is minimizing the error at both t_1 and t_2 (line 8). The training data consists of corresponding velocity fields at t_0 and t_1 . We generate 10 batches of 10 examples, each (lines 9-10), and train the network for 10 epochs (line 11) by iterating over the batches in the data set (line 12) and performing an optimizer step for each (line 13).

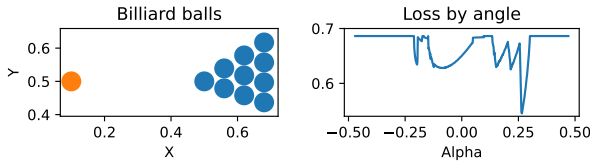
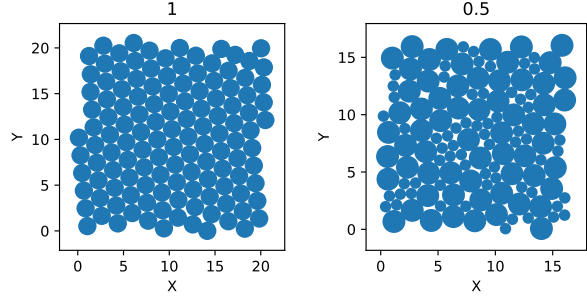


Figure 6. Replication of the billiards experiment from (Hu et al., 2019). Setup with orange cue ball left, and loss $L(\alpha)$ right, where α denotes an angle of the cue ball velocity v_{cue} .

the triangle reaches a desired location after some time. The objective is measured as the mean squared error between desired and observed ball position x , $L(v_{\text{cue}}) = \|x(v_{\text{cue}}) - x^*\|_2^2$. What makes this task challenging is the chaotic nature of the collisions, resulting in discontinuous gradients $\frac{\partial L}{\partial v_{\text{cue}}}$. The corresponding loss landscape depending on the incident angle, $L(\alpha)$ with $\alpha = \tan^{-1}(\frac{v_y}{v_x})$, is shown in Fig. 6.

The original implementation uses a custom Python-like programming language designed for differentiable simulations. To generate an animation of the simulation and the loss plot, it requires 79 lines for computation, and 41 lines for plotting. Our replication consists of 29 lines for computation and 2 lines for plotting, one for the animation and one for the loss graph. We provide our source code for this experiment in the SI.



```

1 from phi.tf.flow import *
2 def energy(x: Tensor, boundary=PERIODIC):
3     dx = boundary.shortest_distance(x, rename_dims(x, 'spheres', 'o'), size)
4     dr = math.vec_length(dx, eps=1e-8) / (R + rename_dims(R, 'spheres', 'o'))
5     return math.l2_loss(math.where((dr < 2e-4) | (dr > 1), 0, 1 - dr))
6 R = wrap([1]*64 + [vec(batch('d'), 1, .5)]*64, instance('spheres'))
7 size = (math.sum(Sphere(vec(x=0, y=0), R).volume, 'spheres') * 1.05) ** .5
8 x0 = math.random_uniform(instance(R), channel('vector'=x,y'), high=size)
9 x_packed = minimize(energy, Solve('L-BFGS-B', x0=x0)) % size
    
```

Figure 7. Replication of the sphere packing experiment from (Schoenholz & Cubuk, 2020). We initialize the system by randomly scattering two types of spheres (line 8), half with radius 1 and half with radius $D \in \{1, 0.5\}$ (line 6). To reach the maximally frustrated state, we minimize the energy (lines 2-5) using L-BFGS-B (line 9). The energy considers all pair-wise distances on the periodic domain (lines 3-4) and pushes overlapping circles apart (line 5). The top plot was generated via `plot(Sphere(x.packed, R), size=(6, 3))`.

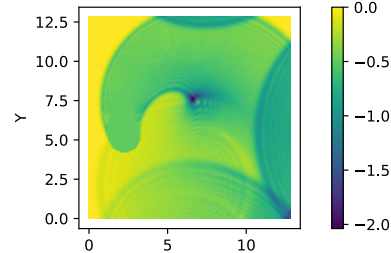


Figure 8. Replication of the wave propagation example from warp (Macklin, 2022).

Sphere packing The goal of this experiment is to pack spheres in a periodic 2D domain with minimal overlap, i.e. to find the maximally frustrated state. This can be achieved by defining an energy function $E(x, R)$ given the positions x and radii R of all particles and then minimizing the energy. As in the original experiment, we divide the spheres into two types, half with radius $R = 1$, the other half with $R = D$, $D \in (0, 1]$. Fig. 7 shows the maximally frustrated states for two values of D , along with our source code. For this result, the original experiment used 25 lines of imports, 25 lines for computation, and 33 lines for plotting.

Wave propagation This example from warp (Macklin, 2022) consists of a wave simulation on a 128×128 grid, driven by a circular obstacle that moves inside the domain. Fig. 8 shows the final wave height after five seconds. Our

implementation, shown in appendix B, consists of 14 lines of code, which is significantly shorter than the original warp implementation at 165 lines.

4. Conclusions

We have extensively demonstrated the capabilities of Φ_{Flow} , both for challenging problems involving PDEs and for general inverse problems. We have showcased the modular nature of Φ_{Flow} as well as its tight integration with the ML libraries PyTorch, Jax and TensorFlow, which can be interchanged by simply modifying the `import` statement. Φ_{Flow} provides a large number of simulation-related functions and unique features, such as automatic matrix generation or the ability to write dimension-agnostic code. The source code provided in this document demonstrates that user code written against Φ_{Flow} 's API is concise and expressive, without sacrificing flexibility or performance. We believe code readability is a greatly underappreciated aspect of machine learning research, resulting in many unnecessary re-implementations, and we hope that Φ_{Flow} will aid in that regard.

Impact Statement

The presented library aims to make it easier for researchers to work with differentiable simulations. We believe that users of Φ_{Flow} will write more interpretable code. We see no harmful impacts of this work.

References

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.
- Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danielka, I., Fantacci, C., Godwin, J., Jones, C., Hennigan, T., Hessel, M., Kapturowski, S., Keck, T., Kemaev, I., King, M., Martens, L., Mikulik, V., Norman, T., Quan, J., Papamakarios, G., Ring, R., Ruiz, F., Sanchez, A., Schneider, R., Sezener, E., Spencer, S., Srinivasan, S., Stokowiec, W., and Viola, F. The DeepMind JAX Ecosystem, 2020. URL <http://github.com/deepmind>.
- Bardenhagen, S. G., Kober, E. M., et al. The generalized interpolation material point method. *Computer Modeling in Engineering and Sciences*, 5(6):477–496, 2004.
- Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
- Bezgin, D. A., Buhendwa, A. B., and Adams, N. A. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications*, pp. 108527, 2022. ISSN 0010-4655. doi: <https://doi.org/10.1016/j.cpc.2022.108527>. URL <https://www.sciencedirect.com/science/article/pii/S0010465522002466>.
- Bieker, K., Peitz, S., Brunton, S. L., Kutz, J. N., and Dellnitz, M. Deep model predictive flow control with limited sensor data and online learning. *Theoretical and computational fluid dynamics*, 34(4):577–591, 2020.
- Bokil, G., Geyer, T., and Wolff, S. Towards convolutional neural networks for heat exchangers in electrified aircraft. 2023.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018a. URL <http://github.com/google/jax>.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018b. URL <http://github.com/google/jax>.
- Brahmachary, S. and Thuerey, N. Unsteady cylinder wakes from arbitrary bodies with differentiable physics-assisted neural network. *arXiv preprint arXiv:2308.04296*, 2023.
- Brandstetter, J., Worrall, D. E., and Welling, M. Message passing neural pde solvers. In *International Conference on Learning Representations*, 2021.
- Brandstetter, J., van den Berg, R., Welling, M., and Gupta, J. K. Clifford neural layers for pde modeling. In *The Eleventh International Conference on Learning Representations*, 2022.
- et al., T. C. Dlpack: Open in memory tensor structure. <https://github.com/dmlc/dlpack>, 2017.
- Ferziger, J. H., Perić, M., and Street, R. L. *Computational methods for fluid dynamics*. springer, 2019.
- Greydanus, S., Dzamba, M., and Yosinski, J. Hamiltonian neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.
- Gupta, J. K. and Brandstetter, J. Towards multi-spatiotemporal-scale generalized pde modeling. *arXiv preprint arXiv:2209.15616*, 2022.

- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL <https://doi.org/10.1038/s41586-020-2649-2>.
- Hennigan, T., Cai, T., Norman, T., and Babuschkin, I. Haiku: Sonnet for JAX, 2020. URL <http://github.com/deepmind/dm-haiku>.
- Holl, P. and Thuerey, N. ϕ -ml: Intuitive scientific computing with dimension types for jax, pytorch, tensorflow & numpy. *Journal of Open Source Software*, 9(95):6171, 2024. doi: 10.21105/joss.06171. URL <https://doi.org/10.21105/joss.06171>.
- Holl, P., Koltun, V., and Thuerey, N. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.
- Holl, P., Koltun, V., and Thuerey, N. Physical gradients for deep learning. *arXiv preprint arXiv:2109.15048*, 2021.
- Holl, P., Koltun, V., and Thuerey, N. Scale-invariant learning by physics inversion. *Advances in Neural Information Processing Systems*, 35:5390–5403, 2022.
- Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. DiffTaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.
- Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.
- Inc., P. T. Collaborative data science, 2015. URL <https://plot.ly>.
- Jameson, A. Aerodynamic shape optimization using the adjoint method. *Lectures at the Von Karman Institute, Brussels*, 2003.
- Jarny, Y., Ozisik, M., and Bardon, J. A general optimization method using adjoint equation for solving multidimensional inverse heat conduction. *International journal of heat and mass transfer*, 34(11):2911–2919, 1991.
- Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021a. ISSN 0027-8424. doi: 10.1073/pnas.2101784118. URL <https://www.pnas.org/content/118/21/e2101784118>.
- Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021b.
- Kurz, J. A., Seman, M. G., Khan, T., Bowman, B. A., and Oian, C. A. Machine learning for pac1d and sese. 2023.
- Li, Z., Patil, S., Shu, D., and Farimani, A. B. Latent neural pde solver for time-dependent systems. In *NeurIPS 2023 AI for Science Workshop*, 2023.
- Macklin, M. Warp: A high-performance python framework for gpu simulation and graphics. <https://github.com/nvidia/warp>, March 2022. NVIDIA GPU Technology Conference (GTC).
- Mitusch, S. K., Funke, S. W., and Dokken, J. S. dolfin-adjoint 2018.1: automated adjoints for fenics and fire-drake. *Journal of Open Source Software*, 4(38):1292, 2019.
- Parekh, N., Zou, A., Jungling, I., Endlich, K., Sadowski, J., and Steinhausen, M. Sex differences in control of renal outer medullary circulation in rats: role of prostaglandins. *American Journal of Physiology-Renal Physiology*, 264(4):F629–F636, 1993.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- Plessix, R.-E. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*, 167(2): 495–503, 2006.
- Raissi, M., Perdikaris, P., and Karniadakis, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.
- Ramos, B., Trost, F., and Thuerey, N. Control of two-way coupled fluid systems with differentiable solvers. In *ICLR 2022 Workshop on Generalizable Policy Learning in Physical World*, 2022.
- Rasp, S. and Thuerey, N. Data-driven medium-range weather prediction with a resnet pretrained on climate simulations: A new model for weatherbench. *Journal of Advances in Modeling Earth Systems*, 13(2): e2020MS002405, 2021.

- Reddy, J. N. *Introduction to the finite element method*. McGraw-Hill Education, 2019.
- Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.
- Runge, C. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.
- Saad, Y. Iterative methods for sparse linear systems. *IEEE Computational Science and Engineering*, 3(4):87, 1996.
- Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pp. 8459–8468. PMLR, 2020.
- Schnell, P., Holl, P., and Thuerey, N. Half-inverse gradients for physical deep learning. *arXiv preprint arXiv:2203.10131*, 2022.
- Schoenholz, S. and Cubuk, E. D. Jax md: a framework for differentiable physics. *Advances in Neural Information Processing Systems*, 33:11428–11441, 2020.
- Sengar, V., Seemakurthy, K., Gubbi, J., and P, B. Multi-task learning based approach for surgical video desmoking. In *Proceedings of the twelfth Indian conference on computer vision, graphics and image processing*, pp. 1–9, 2021.
- Shewchuk, J. R. et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.
- Stachenfeld, K., Fielding, D. B., Kochkov, D., Cranmer, M., Pfaff, T., Godwin, J., Cui, C., Ho, S., Battaglia, P., and Sanchez-Gonzalez, A. Learned coarse models for efficient turbulence simulation. *arXiv preprint arXiv:2112.15275*, 2021.
- Takamoto, M., Praditia, T., Leiteritz, R., MacKinlay, D., Alesiani, F., Pflüger, D., and Niepert, M. Pdebench: An extensive benchmark for scientific machine learning. *Advances in Neural Information Processing Systems*, 35: 1596–1611, 2022.
- Takamoto, M., Alesiani, F., and Niepert, M. Learning neural pde solvers with parameter-guided channel attention. *arXiv preprint arXiv:2304.14118*, 2023.
- Tathawadekar, N., Doan, N. A. K., Silva, C. F., and Thuerey, N. Hybrid neural network pde solvers for reacting flows. *arXiv preprint arXiv:2111.11185*, 2021.
- Teikmanis, O., Leja, L., and Freivalds, K. Applying a differentiable physics simulation to move objects with fluid streams.
- Thapa, S., Li, N., and Ye, J. Dynamic fluid surface reconstruction using deep neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21–30, 2020.
- Thuerey, N., Holl, P., Mueller, M., Schnell, P., Trost, F., and Um, K. Physics-based deep learning. *arXiv preprint arXiv:2109.05237*, 2021.
- Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.
- Tompson, J., Schlachter, K., Sprechmann, P., and Perlin, K. Accelerating eulerian fluid simulation with convolutional networks. In *International Conference on Machine Learning*, pp. 3424–3433. PMLR, 2017.
- Um, K., Brand, R., Fei, Y. R., Holl, P., and Thuerey, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, 33:6111–6122, 2020.
- Van Rossum, G. and Drake Jr, F. L. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.
- Versteeg, H. K. and Malalasekera, W. *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.
- Wandel, N., Weinmann, M., and Klein, R. Learning incompressible fluid dynamics from scratch-towards fast, differentiable fluid models that generalize. In *International Conference on Learning Representations*, 2020a.
- Wandel, N., Weinmann, M., and Klein, R. Teaching the incompressible navier-stokes equations to fast neural surrogate models in 3d. *arXiv preprint arXiv:2012.11893*, 2020b.
- Wandel, N., Weinmann, M., and Klein, R. Teaching the incompressible navier–stokes equations to fast neural surrogate models in three dimensions. *Physics of Fluids*, 33(4), 2021.
- Wang, P. The applications of generative adversarial network in surgical videos. In *Third International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI 2022)*, volume 12509, pp. 300–305. SPIE, 2023.
- Wang, R., Walters, R., and Yu, R. Approximately equivariant networks for imperfectly symmetric dynamics. In *International Conference on Machine Learning*, pp. 23078–23091. PMLR, 2022a.

- Wang, R., Walters, R., and Yu, R. Meta-learning dynamics forecasting using task inference. *Advances in Neural Information Processing Systems*, 35:21640–21653, 2022b.
- Wu, T., Maruyama, T., and Leskovec, J. Learning to accelerate partial differential equations via latent global evolution. *Advances in Neural Information Processing Systems*, 35:2240–2253, 2022.
- Yadav, V. C. et al. *Simulation of Fluid Flows based on the Data-driven Evolution of Vortex Particles*. PhD thesis, Institut für Strömungsmechanik, 2021.

A. Performance measurements

We benchmark all experiments with the three supported machine learning backends: PyTorch, TensorFlow and JAX. We always enable just-in-time (JIT) compilation using Φ_{Flow} 's `@jit_compile` function decorator. The results are shown in Tab. 1. Overall, the performance gap between the backends is reasonably small, and no library consistently outperforms the others. For fluids and tasks involving random data access, JAX usually yields the best performance, while PyTorch works best for easy-to-parallelize tasks.

Table 1. Performance measurements of our experiments by ML backend. The table shows wall-clock time in ms per step on an NVIDIA RTX 3090 excluding warm-up.

	PyTorch	TensorFlow	JAX
Thermal conductivity	24.4 ± 1.5	28.2 ± 1.1	40.5 ± 3.1
PIV	25 ± 2	41.6 ± 1.7	46 ± 2
Learning fluids	293 ± 363	296 ± 356	156 ± 373
Billiards	0.88 ± 1.80	1.20 ± 2.45	0.85 ± 0.21
Sphere packing	2.0 ± 0.3	4.4 ± 0.4	7.8 ± 0.7
2D Waves	0.50 ± 0.07	0.74 ± 0.05	0.38 ± 0.05

B. Code readability

To measure the readability of our code, we prompt ChatGPT with the input "Explain this code to me:", followed by the source code. We show the first output generated by ChatGPT 3.5 for each experiment in full below. We made no changes to our code after reading the explanations generated by ChatGPT to prevent biasing the results.

Sphere packing (Φ_{Flow}) ChatGPT accurately explains our code, recognizing our library PhiFlow and the TensorFlow backend. Its summary is also correct, but it misses the physical interpretation of the energy function. ChatGPT breaks the code into sections and explains each variable separately, citing the appropriate snippets of code. The explanation resembles our description above, which was written without AI-assistance.

Sphere packing (JAX-MD) Given the original Jax-MD code, ChatGPT summarizes the task as "self-assembly of colloidal particles", which matches the domain of the Jax-MD library but is not correct for the given code, as the spheres are explicitly referred to as *bubbles* in the original experiment and code. ChatGPT also fails to recognize the Jax-MD library, despite 7 lines of corresponding imports. Instead, it assumes the code uses JAX (Bradbury et al., 2018a) directly. ChatGPT breaks the code down into nine sections, each with two to seven subsections. It does not cite lines from the source, likely due to the lengthy input.

Billiards (Φ_{Flow}) ChatGPT's explanation of our code is concise and to the point, structuring the code into four sections with three to five points each. Despite the brevity, the output captures all essential parts with sufficient detail, such as the triangular initialization, how the simulation works, and what the loss function is intended to do. ChatGPT even draws on its physics knowledge to explain that the elasticity is connected to the energy conserved in collisions. The output is generally high-level, focusing more on the purpose than on individual variables. All explanations are correct except for two minor issues: (i) ChatGPT did not recognize that only the cue ball velocity is optimized and (ii) it refers to a "controllable ball", which does not match the code. ChatGPT's summary at the bottom perfectly describes the experiment.

Billiards (DiffTaichi) ChatGPT's explanation of the original source code differs greatly from our version. ChatGPT structures the original source code from DiffTaichi into 13 sections, many of which only sport a single bullet point. This lack of structure is likely due to the fact that ChatGPT tries to mirror the order in which variables and functions are defined in the source code. Unlike with our code, the explanations here are low-level, usually limited to individual variables or functions. The bigger picture, such as the purpose of functions or the code as a whole, is not mentioned. Except for missclassifying `sys` and `os` as scientific computing libraries and describing the loss as a "scalar field", the output is mostly correct. However, most statements seem trivial and redundant, e.g. "`collide_pair(t, i, j)`: Function to handle collisions between pairs of balls" or "`vis_resolution`: Resolution for visualization". How the simulation works is not explained; ChatGPT only states that the simulation "initializes the system, performs collision handling, and updates the positions and velocities

of the particles”. The initial triangle configuration as well as the loss function are not explained at all. Evidently ChatGPT understands that the code runs a Billiards simulation but fails to figure out how the simulation works or what its purpose is.

Waves (Φ_{Flow}) ChatGPT identifies the code as a 2D grid simulation of propagating waves, quoting many important lines from the source code. It correctly identifies the purpose of both functions and all of their arguments, and it extracts the resolution, domain size and boundary conditions of the simulation. While it correctly identifies that 60 steps per second with 16 sub-steps each are used, it fails to derive the total simulation time, mixing up seconds and minutes.

Waves (Warp) ChatGPT identifies the code as a grid-based wave propagation simulation but does not extract its resolution, domain size, boundary conditions, or simulation time. No line from the source code is quoted, instead the explanation focusses on the code structure, likely due to the lengthy input. The helper functions `sample`, `laplacian` and `grid_update` are summarized very briefly, and ChatGPT seems unaware that, as kernel functions, these will run on the whole grid in parallel. The descriptions of the main functions `wave_displace` and `wave_solve` are also very short but also partly inaccurate. ChatGPT describes `wave_displace` as displacing points in a circular pattern to simulate wave generation. While technically true, the function simply fills a circle on the grid with a constant value. Furthermore, `wave_solve` is described as solving the wave equation iteratively for each point in the grid. This fails to capture that the function only performs one simulation sub-step, and it misses that the execution is performed in parallel, not iteratively. The bulk of ChatGPT’s response deals with the code structure, such as the `Example` class. Only section 2 out of six sections describes the simulation, and no concrete numbers are given.

The full ChatGPT generations are shown on the next pages, followed by the source code we gave to ChatGPT. For both experiments, the outputs based on our code seem easier to understand, more concise and more insightful.

Sphere Packing (Φ_{Flow})

You: Explain this code to me: `from phi.tf.flow import * [...]`

ChatGPT:

This code appears to be a Python script using the TensorFlow-based library called “PhiFlow” to simulate a physical system involving spheres. Let’s break down the code step by step:

- Import necessary modules:

```
from phi.tf.flow import *
```

This line imports the required modules from the “phi.tf.flow” package. PhiFlow is a library for differentiable physics in TensorFlow. - Define an energy function:

```
def energy(x: Tensor, boundary=PERIODIC):  
    dx = boundary.shortest_distance(x, rename_dims(x, 'spheres', 'o'), size)  
    dr = math.vec_length(dx, eps=1e-8) / (R + rename_dims(R, 'spheres', 'o'))  
    return math.l2_loss(math.where((dr < 2e-4) | (dr > 1), 0, 1 - dr))
```

This function `energy` calculates the energy of a system of spheres at positions `x`. It uses the periodic boundary condition (PERIODIC). The energy is based on the shortest distances (`dx`) between the spheres, and `dr` is a normalized distance. The energy is calculated using the L2 loss of a modified distance. - Set up initial parameters:

```
R = wrap([1]*64 + [vec(batch('d'), 1, .5)]*64, instance('spheres'))
```

This line sets up a list of radii (`R`) for 64 small spheres with radius 1 and 64 larger spheres with radius 0.5. - Compute system size:

```
size = (math.sum(Sphere(vec(x=0, y=0), R).volume, 'spheres') * 1.05) ** .5
```

This line calculates the total volume of the spheres and computes the size of the simulation domain based on that volume. - Initialize particle positions randomly:

```
x0 = math.random_uniform(instance(R), channel(vector='x,y'), high=size)
```

This line initializes the positions (`x0`) of the spheres randomly within the simulation domain. - Minimize the energy using L-BFGS-B optimization:

```
x_packed = minimize(energy, Solve('L-BFGS-B', x0=x0)) % size
```

The code uses the `minimize` function to find the configuration of sphere positions (`x_packed`) that minimizes the energy. The optimization is performed using the L-BFGS-B algorithm. The result is wrapped around to ensure periodic boundary conditions.

In summary, this code sets up a physical system with spheres, defines an energy function based on sphere positions, initializes the system, and then minimizes the energy using an optimization algorithm to find a stable configuration for the spheres. The periodic boundary conditions ensure that the system behaves consistently at the edges of the simulation domain.

Sphere Packing (Φ_{Flow})

```
from phi.tf.flow import *

def energy(x: Tensor, boundary=PERIODIC):
    dx = boundary.shortest_distance(x, rename_dims(x, 'spheres', 'o'), size)
    dr = math.vec_length(dx, eps=1e-8) / (R + rename_dims(R, 'spheres', 'o'))
    return math.l2_loss(math.where((dr < 2e-4) | (dr > 1), 0, 1 - dr))

R = wrap([1]*64 + [vec(batch('d'), 1, .5)]*64, instance('spheres'))
size = (math.sum(Sphere(vec(x=0, y=0), R).volume, 'spheres') * 1.05) ** .5
x0 = math.random_uniform(instance(R), channel(vector='x,y'), high=size)
x_packed = minimize(energy, Solve('L-BFGS-B', x0=x0)) % size
plot(Sphere(x_packed, R), size=(6, 3))
```


Sphere Packing (JAX-MD)

You: Explain this code to me: `import time [...] finalize_plot((2.0, 1))`

ChatGPT:

This code is a Python script that uses JAX (Just Another XLA) to simulate the self-assembly of colloidal particles. The simulation is based on the minimization of a soft sphere pair potential energy function.

Let's break down the code into sections:

- **Importing Libraries:**
 - The code starts by importing necessary libraries like JAX, NumPy, and Matplotlib.
 - It also includes some utilities for progress bars and plot formatting.
- **Progress Bars and Plot Settings:**
 - The `ProgressIter` and `progress` functions are used for displaying progress bars.
 - Matplotlib settings are configured for plot formatting.
- **System Visualization Functions:**
 - `draw_system`: Function to visualize the system by plotting particle positions.
 - `finalize_plot`: Function to adjust the plot size and layout.
- **System Property Utilities:**
 - `box_size_at_number_density`: Calculates the box size based on particle count and number density.
- **Simulation Parameters:**
 - `N`: Total number of particles.
 - `simulation_steps`: Number of simulation steps.
 - `dt`: Time step for the simulation.
 - `temperature`: Temperature of the system.
 - `key`: Random key for initializing the system.
 - `packing_fraction`: Desired packing fraction of particles in the system.
 - `markersize`: Size of markers for particle visualization.
- **Particle Species and Interaction:**
 - The system consists of two types of particles, species 0 and species 1.
 - `species_sigma`: Function to define the diameter matrix based on particle species.
- **Simulation Function:**
 - `simulation`: Main simulation function that takes a diameter and a random key as input.
 - It sets up the simulation environment, defines the soft sphere pair potential energy, initializes the system, and performs minimization steps.
- **Simulation Visualization:**
 - The script plots two visualizations using Matplotlib:
 - The first subplot shows the self-assembly of particles with a diameter of 1.0.
 - The second subplot shows the self-assembly of particles with diameters 0.8 and 1.0.
 - **Plotting and Finalization:**
 - The `draw_system` function is used to plot the particle positions.
 - The `finalize_plot` function adjusts the plot size and layout.

In summary, the code simulates the self-assembly of colloidal particles using JAX and visualizes the results with Matplotlib. The particles interact through a soft sphere pair potential energy, and the system's evolution is visualized at different diameters.

Sphere Packing (JAX-MD)

```
import time

from functools import partial
import numpy as onp

import jax.numpy as np

from jax.api import jit
from jax.api import grad
from jax.api import vmap
from jax.api import value_and_grad

from jax import random
from jax import lax

from jax.experimental import stax
from jax.experimental import optimizers

from jax.config import config
config.update('jax_enable_x64', True)

from jax_md import space
from jax_md import minimize
from jax_md import simulate
from jax_md import space
from jax_md import energy
from jax_md import quantity
from jax_md.util import f32, f64

# Progress Bars.
from IPython.display import HTML, display
import time

def ProgressIter(iter_fun, iter_len=0):
    if not iter_len:
        iter_len = len(iter_fun)
    out = display(progress(0, iter_len), display_id=True)
    for i, it in enumerate(iter_fun):
        yield it
        out.update(progress(i + 1, iter_len))

def progress(value, max):
    return HTML("""
    <progress
      value='{value}'
      max='{max}',
      style='width: 45%'
    >
      {value}
    </progress>
    """).format(value=value, max=max)

# Matplotlib settings.

import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style(style='white')
sns.set(font_scale=1.6)

def format_plot(x, y):
    plt.xlabel(x, fontsize=20)
    plt.ylabel(y, fontsize=20)

def finalize_plot(shape=(1, 1)):
    plt.gcf().set_size_inches(
        shape[0] * 1.5 * plt.gcf().get_size_inches()[1],
        shape[1] * 1.5 * plt.gcf().get_size_inches()[1])
    plt.tight_layout()

def draw_system(R, box_size, marker_size, color=None):
    if color == None:
        color = [64 / 256] * 3
    ms = marker_size / box_size

    R = onp.array(R)

    marker_style = dict(
        linestyle='none',
        markeredgewidth=3,
        marker='o',
        markersize=ms,
        color=color,
        fillstyle='none')

    plt.plot(R[:, 0], R[:, 1], **marker_style)
```

```

plt.plot(R[:, 0] + box_size, R[:, 1], **marker_style)
plt.plot(R[:, 0], R[:, 1] + box_size, **marker_style)
plt.plot(R[:, 0] + box_size, R[:, 1] + box_size, **marker_style)
plt.plot(R[:, 0] - box_size, R[:, 1], **marker_style)
plt.plot(R[:, 0], R[:, 1] - box_size, **marker_style)
plt.plot(R[:, 0] - box_size, R[:, 1] - box_size, **marker_style)

plt.xlim([0, box_size])
plt.ylim([0, box_size])
plt.axis('off')
plt.gca().set_facecolor([1, 1, 1])

# System property utilities.

def box_size_at_number_density(particle_count, number_density):
    return f32((particle_count / number_density) ** 0.5)

N = 128
N_2 = N // 2

simulation_steps = np.arange(300)
dt = 1e-2
temperature = 1e-5

key = random.PRNGKey(0)

packing_fraction = 0.98
markersize = 260

def box_size_at_packing_fraction(diameter):
    bubble_volume = N_2 * np.pi * (diameter ** 2 + 1) / 4
    return np.sqrt(bubble_volume / packing_fraction)

species = np.array([0] * (N_2) + [1] * (N_2), dtype=np.int32)

def species_sigma(diameter):
    d_AA = diameter
    d_BB = 1
    d_AB = 0.5 * (diameter + 1)
    return np.array(
        [[d_AA, d_AB],
         [d_AB, d_BB]]
    )

@jit
def simulation(diameter, key):
    # Create the simulation environment.
    box_size = box_size_at_packing_fraction(diameter)
    displacement, shift = space.periodic(box_size)

    # Create the energy function.
    sigma = species_sigma(diameter)
    energy_fun = energy.soft_sphere_pair(
        displacement, species=species, sigma=sigma)

    # Randomly initialize the system.
    R = random.uniform(key, (N, 2), maxval=box_size)

    # Minimize to the nearest minimum.
    init_fn, apply_fn = minimize.fire_descent(energy_fun, shift)
    state = init_fn(R)

    do_step = lambda state, t: (apply_fn(state, t=t), ())
    state, _ = lax.scan(do_step, state, simulation_steps)

    return box_size, energy_fun(state.position), state.position

plt.subplot(1, 2, 1)

box_size, raft_energy, bubble_positions = simulation(1.0, key)
draw_system(bubble_positions, box_size, markersize)
finalize_plot((0.5, 0.5))

plt.subplot(1, 2, 2)

box_size, raft_energy, bubble_positions = simulation(0.8, key)
draw_system(bubble_positions[:N_2], box_size, 0.8 * markersize)
draw_system(bubble_positions[N_2:], box_size, markersize)
finalize_plot((2.0, 1))

```

Billiards (Φ_{Flow})

You: Explain this code to me: `from phi.torch.flow import *` [...]

ChatGPT:

This code appears to be a Python script that defines a simulation of billiard balls moving in a triangular configuration. Let's break down the code into its main components:

- **billiards_triangle function:**
 - This function generates the initial configuration of the billiard balls arranged in a triangular pattern.
 - `billiard_layers` determines the number of layers in the triangle, and `radius` is the radius of each ball.
 - The function uses nested loops to create the ball coordinates based on the layer and position within the layer.
- **physics_step function:**
 - This function simulates a physics step for the billiard balls.
 - It uses the `phi` library (specifically, `advect.points` and `math` functions) for numerical simulation.
 - The function calculates the next position of the balls (`v_next`) based on advection (movement).
 - It then computes the distance between balls, checks for collisions, and updates the velocities accordingly.
 - Elasticity determines how much energy is conserved during collisions.
- **loss_function function:**
 - This function defines the loss function for the optimization process.
 - It takes initial position `x0` and initial velocity `v0` as inputs and simulates the motion of the billiard balls using the `physics_step` function.
 - The goal is to minimize the L2 loss between the final position of the controllable ball and a specified goal position.
 - The function returns the L2 loss and the trajectory of the simulation.
- **Last part:**
 - The script sets an initial position `x0` and a function `v0` that generates initial velocities based on an angle `alpha`.
 - It then uses the `loss_function` to create a scan over different values of `alpha`, aiming to find the best initial velocity that minimizes the L2 loss.
 - The result is stored in the `scan` variable, which represents a centered grid of loss values over the specified range of `alpha`.

In summary, this code defines a simulation of billiard balls with a triangular initial configuration, and it aims to find the optimal initial velocity to reach a specified goal position. The optimization is performed over a range of angles using the `phi` library for numerical simulations.

Billiards (Φ_{Flow})

```
from phi.torch.flow import *

def billiards_triangle(billiard_layers=4, radius=.03):
    coords = []
    for i in range(billiard_layers):
        for j in range(i + 1):
            coords.append(vec(x=i * 2 * radius + 0.5, y=j * 2 * radius + 0.5 - i * radius * 0.7))
    return Sphere(stack(coords, instance('balls')), radius=radius)

def physics_step(v: PointCloud, dt: float, elasticity=0.8):
    v_next = advect.points(v, v, dt)
    dist = v_next.points - rename_dims(v_next.points, 'balls', 'others')
    dist_norm = math.vec_length(dist, eps=1e-4) # eps to avoid NaN during backprop of sqrt
    rel_v = v.values - rename_dims(v.values, 'balls', 'others')
    dist_dir = math.safe_div(dist, dist_norm)
    projected_v = dist_dir.vector * rel_v.vector
    has_impact = (projected_v < 0) & (dist_norm < 2 * v.geometry.radius)
    impulse = -(1 + elasticity) * .5 * projected_v * dist_dir
    radius_sum = v.geometry.radius + rename_dims(v.geometry.radius, 'balls', 'others')
    impact_time = math.safe_div(dist_norm - radius_sum, projected_v)
    x_inc_contrib = math.sum(math.where(has_impact, math.minimum(impact_time - dt, 0) * impulse, 0), 'others')
    v = v.with_elements(v.geometry.shifted(x_inc_contrib))
    v += math.sum(math.where(has_impact, impulse, 0), 'others')
    return advect.points(v, v, dt)

def loss_function(x0: Tensor, v0: Tensor, goal=vec(x=0.9, y=0.75), steps=1024):
    triangle_balls = PointCloud(billiards_triangle()) * (0, 0)
    controllable_ball = PointCloud(Sphere(expand(x0, instance(triangle_balls).with_size(1)),
    radius=triangle_balls.geometry.radius)) * v0
    all_balls = controllable_ball & triangle_balls
    trj = iterate(physics_step, batch(t=steps), all_balls, f_kwargs={'dt': 0.003})
    return math.l2_loss(trj.t[-1].balls[-1] - goal), trj

x0 = vec(x=.1, y=.5)
v0 = lambda alpha: vec(x=0.3 * math.cos(alpha), y=0.3 * math.sin(alpha))
scan = CenteredGrid(lambda alpha: loss_function(x0, v0(alpha))[0], alpha=1000, bounds=Box(alpha=(-PI/4, PI/4)))
plot({"Billiard balls": [billiards_triangle(), Sphere(x0, radius=.03)], "Loss by angle": scan.alpha[200:-200]}, size=(6, 1.8),
overlay='list')
```

Billiards (DiffTaichi)

You: Explain this code to me: `import taichi as ti [...] scan(float(sys.argv[1]))`

ChatGPT:

This code is a simulation of billiards using the Taichi programming language. It employs a numerical approach to simulate the physics of billiards, including collision detection and response.

Here is a breakdown of the code:

- **Importing Libraries:**
 - `taichi, sys, math, numpy, os`: Libraries for scientific computing and simulation.
 - `matplotlib.pyplot`: Used for plotting graphs.
- **Taichi Initialization:**
 - `ti.init()`: Initializes the Taichi runtime with a default floating-point precision (real is set to `ti.f32`).
- **Simulation Parameters:**
 - `max_steps, vis_interval, output_vis_interval, steps`: Parameters for the simulation.
 - `vis_resolution`: Resolution for visualization.
 - `scalar` and `vec`: Functions to define scalar and vector fields.
 - `loss`: Scalar field to represent the loss.
- **Particle Properties and Simulation Setup:**
 - `init_x, init_v`: Initial positions and velocities.
 - `x, v, x_inc, impulse`: Particle positions, velocities, increments, and impulses.
 - `billiard_layers, n_balls, target_ball`: Parameters defining the billiard scene.
 - `goal, radius, elasticity`: Goal position, radius of the balls, and elasticity factor.
 - `ti.root`: Defines the hierarchical structure of the simulation data.
- **Simulation Parameters and Constants:**
 - `dt, alpha, learning_rate`: Time step, regularization parameter, and learning rate.
- **Collision Pair Function:**
 - `collide_pair(t, i, j)`: Function to handle collisions between pairs of balls.
- **Collision, Advance, and Loss Computation Kernels:**
 - `collide(t)`: Kernel to handle collisions between all pairs of balls.
 - `advance(t)`: Kernel to update particle velocities and positions based on impulses.
 - `compute_loss(t)`: Kernel to compute the loss at a given time step.
- **Initialization Function:**
 - `initialize()`: Initializes the initial positions and velocities of the particles.
- **Taichi GUI Setup:**
 - `ti.GUI`: Initializes a graphical user interface for visualization.
- **Forward Simulation Function:**

- `forward(visualize=False, output=None)`: Main simulation function that initializes the system, performs collision handling, and updates the positions and velocities of the particles.
- **Clear Function:**
- `clear()`: Resets the impulse and position increment fields.
- **Scan Function:**
- `scan(zoom)`: Performs a parameter scan by varying the initial velocity angle and plots the resulting loss.
- **Simulation Execution:**
- The script executes by calling the `scan` function with a zoom factor specified from the command line argument (`sys.argv[1]`).

The simulation progresses through time steps, handling collisions, updating positions and velocities, and computing the loss. The final loss values are plotted against the angle of the initial velocity, providing insights into the objective of the billiard scene.

Billiards (DiffTaichi)

```
import taichi as ti
import sys
import math
import numpy as np
import os
import matplotlib.pyplot as plt

real = ti.f32
ti.init(default_fp=real, flatten_if=True)

max_steps = 2048
vis_interval = 64
output_vis_interval = 16
steps = 1024
assert steps * 2 <= max_steps

vis_resolution = 1024

scalar = lambda: ti.field(dtype=real)
vec = lambda: ti.Vector.field(2, dtype=real)

loss = scalar()

init_x = vec()
init_v = vec()

x = vec()
x_inc = vec() # for TOI
v = vec()
impulse = vec()

billiard_layers = 4
n_balls = 1 + (1 + billiard_layers) * billiard_layers // 2
target_ball = n_balls - 1
# target_ball = 0
goal = [0.9, 0.75]
radius = 0.03
elasticity = 0.8

ti.root.dense(ti.i, max_steps).dense(ti.j, n_balls).place(x, v, x_inc, impulse)
ti.root.place(init_x, init_v)
ti.root.place(loss)
ti.root.lazy_grad()

dt = 0.003
alpha = 0.00000
learning_rate = 0.01

@ti.func
def collide_pair(t, i, j):
    imp = ti.Vector([0.0, 0.0])
    x_inc_contrib = ti.Vector([0.0, 0.0])
    if i != j:
        dist = (x[t, i] + dt * v[t, i]) - (x[t, j] + dt * v[t, j])
        dist_norm = dist.norm()
        rela_v = v[t, i] - v[t, j]
        if dist_norm < 2 * radius:
            dir = ti.Vector.normalized(dist, 1e-6)
            projected_v = dir.dot(rela_v)

            if projected_v < 0:
                imp = -(1 + elasticity) * 0.5 * projected_v * dir
                toi = (dist_norm - 2 * radius) / min(
                    -1e-3, projected_v) # Time of impact
                x_inc_contrib = min(toi - dt, 0) * imp
    x_inc[t + 1, i] += x_inc_contrib
    impulse[t + 1, i] += imp

@ti.kernel
def collide(t: ti.i32):
    for i in range(n_balls):
        for j in range(i):
            collide_pair(t, i, j)
    for i in range(n_balls):
        for j in range(i + 1, n_balls):
            collide_pair(t, i, j)

@ti.kernel
def advance(t: ti.i32):
    for i in range(n_balls):
        v[t, i] = v[t - 1, i] + impulse[t, i]
        x[t, i] = x[t - 1, i] + dt * v[t, i] + x_inc[t, i]

@ti.kernel
def compute_loss(t: ti.i32):
    loss[None] = (x[t, target_ball][0] - goal[0])**2 + (x[t, target_ball][1] -
```

```
goal[1])**2
```

```
@ti.kernel
def initialize():
    x[0, 0] = init_x[None]
    v[0, 0] = init_v[None]

gui = ti.GUI("Billiards", (1024, 1024), background_color=0x3C733F)

def forward(visualize=False, output=None):
    initialize()

    interval = vis_interval
    if output:
        interval = output_vis_interval
        os.makedirs('billiards/{}'.format(output), exist_ok=True)

    count = 0
    for i in range(billiard_layers):
        for j in range(i + 1):
            count += 1
            x[0, count] = [
                i * 2 * radius + 0.5, j * 2 * radius + 0.5 - i * radius * 0.7
            ]

    pixel_radius = int(radius * 1024) + 1

    for t in range(1, steps):
        collide(t - 1)
        advance(t)

        if (t + 1) % interval == 0 and visualize:
            gui.clear()
            gui.circle((goal[0], goal[1]), 0x000000, pixel_radius // 2)

            for i in range(n_balls):
                if i == 0:
                    color = 0xCCCCCC
                elif i == n_balls - 1:
                    color = 0x3344cc
                else:
                    color = 0xF20530

                gui.circle((x[t, i][0], x[t, i][1]), color, pixel_radius)

            if output:
                gui.show('billiards/{}{:04d}.png'.format(output, t))
            else:
                gui.show()

    compute_loss(steps - 1)

@ti.kernel
def clear():
    for t, i in ti.ndrange(max_steps, n_balls):
        impulse[t, i] = ti.Vector([0.0, 0.0])
        x_inc[t, i] = ti.Vector([0.0, 0.0])

def scan(zoom):
    N = 1000
    angles = []
    losses = []
    forward(visualize=True, output='initial')
    for i in range(N):
        alpha = ((i + 0.5) / N - 0.5) * math.pi * zoom
        init_x[None] = [0.1, 0.5]
        init_v[None] = [0.3 * math.cos(alpha), 0.3 * math.sin(alpha)]

        loss[None] = 0
        clear()
        forward(visualize=False)
        print(loss[None])

        losses.append(loss[None])
        angles.append(math.degrees(alpha))

    plt.plot(angles, losses)
    fig = plt.gcf()
    fig.set_size_inches(5, 3)
    plt.title('Billiard Scene Objective')
    plt.ylabel('Objective')
    plt.xlabel('Angle of velocity')
    plt.tight_layout()
    plt.show()

scan(float(sys.argv[1]))
```

Waves (Φ_{Flow})

You: Explain this code to me: `from phi.jax.flow import * [...]`

ChatGPT:

This code seems to be a simulation of a wave propagating through a 2D grid, possibly representing a water surface or similar phenomenon. Let's break it down step by step:

1. Import Statements:

```
from phi.jax.flow import *
```

This imports necessary functions and classes from the `phi` library for running simulations with computational fluid dynamics (CFD) applications.

2. Function Definitions:

- `wave_displace(sphere: Sphere, *fields: Field, mag=.5, t=-math.PI * 0.5)`: This function takes a `Sphere` object and one or more `Field` objects as input. It displaces the fields based on the distance to the sphere and a time-dependent displacement value. The displacement value is calculated as a sinusoidal function of time (`t`). It returns a list of displaced fields.
- `step(h_c, h_p, time, dt=1/60./16, k_speed=1.0, k_damp=0.0)`: This function defines a single time step in the simulation. It takes the current state of the simulation grid `h_c`, the previous state `h_p`, the current time, and optional parameters for time step (`dt`), wave speed coefficient (`k_speed`), and damping coefficient (`k_damp`). It computes the next state `h_n` based on the wave equation and returns it along with updated values of the current grid and time.

3. Initial Conditions:

```
h_initial = CenteredGrid(x=128, y=128, bounds=Box(x=12.8, y=12.8),  
boundary=ZERO_GRADIENT)
```

This initializes the simulation grid (`h_initial`) with a resolution of 128x128 cells and a spatial domain defined by a bounding box of size 12.8x12.8. `ZERO_GRADIENT` indicates the boundary condition where the gradient at the boundary is assumed to be zero.

4. Simulation Execution:

```
final, *_ = iterate(step, 5 * 60 * 16, h_initial, h_initial, 0)
```

This line executes the simulation. It iterates the `step` function for a total of 5 minutes (using 60 steps per second and 16 iterations per step). It starts with the initial grid state (`h_initial`) and previous state (`h_initial`) and initial time `0`. It returns the final state of the simulation grid.

5. Visualization: `show(final)`

This line displays the final state of the simulation grid.

In summary, this code simulates the propagation of waves through a 2D grid using computational fluid dynamics principles, possibly representing phenomena like water waves, with the capability to displace the grid based on a spherical object.

Waves (Φ_{Flow})

```
from phi.jax.flow import *

def wave_displace(sphere: Sphere, *fields: Field, mag=.5, t=-math.PI * 0.5):
    sq_dist_to_sphere = math.vec_squared(fields[0].center - sphere.center)
    displace_val = mag * math.sin(t)
    return [field.where(sq_dist_to_sphere < sphere.radius**2, displace_val, f) for f in fields]

@jit_compile
def step(h_c, h_p, time, dt=1/60./16, k_speed=1.0, k_damp=0.0):
    sphere = Sphere(center=h_c.bounds.center + math.rotate_vector(vec(x=0, y=-12.8/3), time), radius=1.)
    h_c, h_p = wave_displace(sphere, h_c, h_p)
    h_n = 2.0 * h_c - h_p + dt * dt * (k_speed * h_c.laplace() - k_damp * (h_c - h_p)) # wave_solve
    return h_n, h_c, time + dt

h_initial = CenteredGrid(x=128, y=128, bounds=Box(x=12.8, y=12.8), boundary=ZERO_GRADIENT)
final, *_ = iterate(step, 5 * 60 * 16, h_initial, h_initial, 0)
plot(final)
```

Waves (Warp)

You: Explain this code to me: `import math` [...]

ChatGPT:

This code appears to be a simulation of wave propagation in a grid-based environment, with a visualization component. Let's break it down:

- Imports:** The code imports necessary libraries, including `math` for mathematical operations and a module named `warp` for parallel computation and rendering.
- Function Definitions:**
 - `sample`: Retrieves the value from a 2D array at specified coordinates, clamping the coordinates within the array boundaries.
 - `laplacian`: Computes the Laplacian of a scalar field at a given point in the grid.
 - `wave_displace`: Displaces the grid points around a specified center in a circular pattern to simulate wave generation.
 - `wave_solve`: Solves the wave equation iteratively for each point in the grid.
 - `grid_update`: Updates the vertex positions of a grid based on the calculated heights.
- Class Definition:**
 - `Example`: Manages the simulation and rendering of the wave propagation.
 - Initializes simulation parameters, grid dimensions, and wave constants.
 - Generates grid vertices and indices for rendering.
 - Provides methods to advance the simulation (`step`) and render the current state (`render`).
- Main Execution:**
 - Instantiates an `Example` object.
 - Iterates over simulation frames, advancing the simulation and rendering each frame.
 - If a renderer is available, saves the rendered frames to a file.
- Simulation Process:**
 - In each frame, the simulation advances in discrete substeps.
 - Wave displacement and wave equation solving kernels are launched in parallel for each grid point.
 - Grid heights are updated based on the wave simulation results.
 - The simulation time is incremented.

6. **Rendering:**

- If a renderer is available, the grid vertices along with a sphere representing the wave source are rendered in a 3D scene.
- The rendering process is timed using ScopedTimer.

Overall, the code simulates the propagation of waves in a grid-based environment and visualizes the results using parallel computation and rendering techniques provided by the warp module.

Waves (Warp)

```
import math

import warp as wp
import warp.render

wp.init()

@wp.func
def sample(f: wp.array(dtype=float), x: int, y: int, width: int, height: int):
    # clamp texture coords
    x = wp.clamp(x, 0, width - 1)
    y = wp.clamp(y, 0, height - 1)

    s = f[y * width + x]
    return s

@wp.func
def laplacian(f: wp.array(dtype=float), x: int, y: int, width: int, height: int):
    ddx = sample(f, x + 1, y, width, height) - 2.0 * sample(f, x, y, width, height) + sample(f, x - 1, y, width, height)
    ddy = sample(f, x, y + 1, width, height) - 2.0 * sample(f, x, y, width, height) + sample(f, x, y - 1, width, height)

    return ddx + ddy

@wp.kernel
def wave_displace(
    hcurrent: wp.array(dtype=float),
    hprevious: wp.array(dtype=float),
    width: int,
    height: int,
    center_x: float,
    center_y: float,
    r: float,
    mag: float,
    t: float,
):
    tid = wp.tid()

    x = tid % width
    y = tid // width

    dx = float(x) - center_x
    dy = float(y) - center_y

    dist_sq = float(dx * dx + dy * dy)

    if dist_sq < r * r:
        h = mag * wp.sin(t)

        hcurrent[tid] = h
        hprevious[tid] = h

@wp.kernel
def wave_solve(
    hprevious: wp.array(dtype=float),
    hcurrent: wp.array(dtype=float),
    width: int,
    height: int,
    inv_cell: float,
    k_speed: float,
    k_damp: float,
    dt: float,
):
    tid = wp.tid()

    x = tid % width
    y = tid // width

    l = laplacian(hcurrent, x, y, width, height) * inv_cell * inv_cell

    # integrate
    h1 = hcurrent[tid]
    h0 = hprevious[tid]

    h = 2.0 * h1 - h0 + dt * dt * (k_speed * l - k_damp * (h1 - h0))

    # buffers get swapped each iteration
    hprevious[tid] = h

# simple kernel to apply height deltas to a vertex array
@wp.kernel
def grid_update(heights: wp.array(dtype=float), vertices: wp.array(dtype=wp.vec3)):
    tid = wp.tid()

    h = heights[tid]
    v = vertices[tid]
```

```
v_new = wp.vec3(v[0], h, v[2])
```

```
vertices[tid] = v_new
```

```
class Example:
```

```
def __init__(self, stage):
    self.sim_width = 128
    self.sim_height = 128

    self.sim_fps = 60.0
    self.sim_substeps = 16
    self.sim_duration = 5.0
    self.sim_frames = int(self.sim_duration * self.sim_fps)
    self.sim_dt = (1.0 / self.sim_fps) / self.sim_substeps
    self.sim_time = 0.0

    # wave constants
    self.k_speed = 1.0
    self.k_damp = 0.0

    # grid constants
    self.grid_size = 0.1
    self.grid_displace = 0.5

    vertices = []
    self.indices = []

    def grid_index(x, y, stride):
        return y * stride + x

    for z in range(self.sim_height):
        for x in range(self.sim_width):
            pos = (
                float(x) * self.grid_size,
                0.0,
                float(z) * self.grid_size,
            )

            # directly modifies verts_host memory since this is a numpy alias of the same buffer
            vertices.append(pos)

            if x > 0 and z > 0:
                self.indices.append(grid_index(x - 1, z - 1, self.sim_width))
                self.indices.append(grid_index(x, z, self.sim_width))
                self.indices.append(grid_index(x, z - 1, self.sim_width))

                self.indices.append(grid_index(x - 1, z - 1, self.sim_width))
                self.indices.append(grid_index(x - 1, z, self.sim_width))
                self.indices.append(grid_index(x, z, self.sim_width))

    # simulation grids
    self.sim_grid0 = wp.zeros(self.sim_width * self.sim_height, dtype=float)
    self.sim_grid1 = wp.zeros(self.sim_width * self.sim_height, dtype=float)
    self.sim_verts = wp.array(vertices, dtype=wp.vec3)

    # create surface displacement around a point
    self.cx = self.sim_width / 2 + math.sin(self.sim_time) * self.sim_width / 3
    self.cy = self.sim_height / 2 + math.cos(self.sim_time) * self.sim_height / 3

    self.renderer = None
    if stage:
        self.renderer = wp.render.UsdRenderer(stage)

    def step(self):
        with wp.ScopedTimer("step", active=True):
            for s in range(self.sim_substeps):
                # create surface displacement around a point
                self.cx = self.sim_width / 2 + math.sin(self.sim_time) * self.sim_width / 3
                self.cy = self.sim_height / 2 + math.cos(self.sim_time) * self.sim_height / 3

                wp.launch(
                    kernel=wave_displace,
                    dim=self.sim_width * self.sim_height,
                    inputs=[
                        self.sim_grid0,
                        self.sim_grid1,
                        self.sim_width,
                        self.sim_height,
                        self.cx,
                        self.cy,
                        10.0,
                        self.grid_displace,
                        -math.pi * 0.5,
                    ],
                )

            # integrate wave equation
            wp.launch(
                kernel=wave_solve,
                dim=self.sim_width * self.sim_height,
                inputs=[
                    self.sim_grid0,
                    self.sim_grid1,
                ]
```

```

        self.sim_width,
        self.sim_height,
        1.0 / self.grid_size,
        self.k_speed,
        self.k_damp,
        self.sim_dt,
    ],
)

# swap grids
(self.sim_grid0, self.sim_grid1) = (self.sim_grid1, self.sim_grid0)

self.sim_time += self.sim_dt

with wp.ScopedTimer("mesh", active=False):
    # update grid vertices from heights
    wp.launch(kernel=grid_update, dim=self.sim_width * self.sim_height, inputs=[self.sim_grid0, self.sim_verts])

def render(self):
    if self.renderer is None:
        return

    with wp.ScopedTimer("render", active=True):
        vertices = self.sim_verts.numpy()

        self.renderer.begin_frame(self.sim_time)
        self.renderer.render_mesh("surface", vertices, self.indices, colors=((0.35, 0.55, 0.9),) * len(vertices))
        self.renderer.render_sphere(
            "sphere",
            (self.cx * self.grid_size, 0.0, self.cy * self.grid_size),
            (0.0, 0.0, 0.0, 1.0),
            10.0 * self.grid_size,
            color=(1.0, 1.0, 1.0),
        )
        self.renderer.end_frame()

if __name__ == "__main__":
    stage_path = "example_wave.usd"

    example = Example(stage_path)

    for i in range(example.sim_frames):
        example.step()
        example.render()

    if example.renderer:
        example.renderer.save()

```