Φ_{Flow}: Differentiable Simulations for Machine Learning

Philipp Holl [1] **Nils Thuerey** [1]

Abstract

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 ML 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} is available at `https://github.com/tum-pbs/PhiFlow`.

1. Introduction

Inferring physical parameters or states with neural networks has sparked a multitude of promising lines of research. Neural networks have been used to accelerate simulations [1], [2], improve simulation accuracy for fixed resolutions [3], [4], control complex physical systems [5], [6], encode physical states and sequences [7] and find conservation laws [8], among others. All of these tasks can be learned end-to-end only with the aid of a differentiable simulator, as the network predictions pass through a simulator in the forward pass. However, many differentiable simulators don't allow for seamless integration with ML frameworks, forcing users to manually handle the forward and backward data flow [9]–[11]. Meanwhile, the ML libraries PyTorch [12], TensorFlow [13], and Jax [14]–[16] lack vital functionality required for many types of simulations, which has led researchers to either not use end-to-end training [1], [17]–[19] or implement custom differentiable simulations [2], [3], [5] or libraries [11], [20]–[23] compatible with only one ML framework, preventing adoption in the other communities.

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 [24]. It is intended to be used for a wide variety of simulations and includes high-level data structures for grid/mesh-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. It has been used in production for multiple works and publications [4], [6], [25]–[48], as well as open data sets [49], [50].

2. Major features of Φ_{Flow}

All of Φ_{Flow}'s core functionality is implemented directly in Python 3 [51]. This makes it easy to understand its source code and enables seamless integration with PyTorch, TensorFlow and Jax using the abstraction layer Φ_{ML} [52] which supports DLPack [53]. Φ_{Flow} provides classes to represent grids, graphs, unstructured meshes, point clouds, as well as various primitive geometries. Next we present features that Φ_{Flow} adds on top of the ML libraries with further details in appendix A.

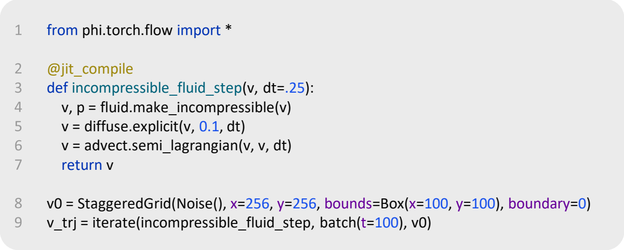
Fluid solver Incompressible fluid simulations are challenging, both theoretically and numerically [54]. Unlike many other toolkits, Φ_{Flow} does not provide a stand-alone solver but rather a set of building blocks from which a full simulation can quickly be assembled as in Fig. 1. These include functions for advection, diffusion, and pressure computation, giving users full control and making it easy to adapt the code to their specific needs.

Discrete differential operators 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

---

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

Published at the 2nd Differentiable Almost Everything Workshop at the 41st International Conference on Machine Learning, Vienna, Austria. July 2024. Copyright 2024 by the author(s).



```
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 1. Executable source code of an incompressible fluid simulation using staggered grids running on PyTorch.

higher-order operators, letting users specify arbitrary spatial orders of accuracy for which the stencils are generated on-the-fly.

Boundary conditions Φ_{Flow} provides an extensive library dedicated to boundary condition (BC) handling, including Dirichlet, Neumann, periodic and symmetric BCs, all of which support BC arithmetic, padding and distance functions, as well as index transforms. All built-in physics functions are coded to work with all of these BCs and adjust the employed numerical scheme accordingly.

Dimensionality-agnostic code Many PDEs can be realized in multiple dimensionalities (1D, 2D, 3D), because their mathematical formulations are abstract, e.g. $\dot{u} = \kappa \nabla^2 u$. We introduce a system for writing dimensionality-agnostic code which works by letting users mark tensor dimensions as being *spatial*. All relevant functions then infer the dimensionality from the number of tagged dimensions. The code given in Fig. 1 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 Φ_{Flow} includes a novel system of controlling precision that is more predictable, less error-prone and easier to control than existing solutions. All operations determine the desired precision from the operation context rather than the data types of its inputs. The precision can be set globally or locally, and operations automatically convert non-matching tensors, avoiding all data type errors. To enable double precision in Fig. 1, insert `math.set_global_precision(64)` below line 1.

Fully differentiable preconditioned linear solvers Φ_{Flow} includes all SciPy solvers, as well as custom GPU-compatible conjugate gradient and (stabilized) bi-conjugate gradient methods [55] 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 [56] and clustering, which can drastically improve convergence speed. We support differentiating not only w.r.t. the right-hand-side but also the (sparse) matrix and all of its dependencies, a feature that is missing from the base ML libraries.

Automatic matrix generation via function tracing Given an affine function $\hat{A}(x)$, Φ_{Flow} can build a sparse matrix A and offset o, such that $\hat{A}(x) = A \cdot x + o$. This allows users to conveniently express linear systems of equations as interpretable Python functions but solve them with the efficiency of explicit representations. This is achieved by tracing low-level affine operations, similar to just-in-time (JIT) compilation. The matrix assembly itself can be JIT compiled to avoid tracing overheads at runtime.

Integration of SciPy optimizers Φ_{Flow} includes a simple API to use all SciPy optimizers, integrating them into the computational graph of the base ML library. 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.

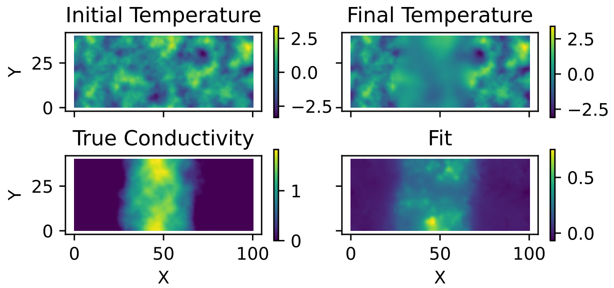
Vectorization via batch dimensions Φ_{Flow} supports parallelization via batch dimensions for *all* operations. This eliminates the need for a `vmap` function which would prevent debugging or visualizing the parallelized code. In Φ_{Flow}, dimensions can be tagged as *batch*, allowing users to define arbitrarily many batch dimensions. Quantities with different batch dimensions are automatically reshaped to match, making user code parallelize trivially. Specifying `batch(config=n)` in line 8 of Fig. 1 runs n parallel fluid simulations with different initial conditions.

Single-call visualization Φ_{Flow} includes a plotting frontend for Matplotlib [57] and Plotly [58] that creates fully-fledged figures from a single `plot()` call. Despite this simplicity, the above-mentioned dimension tags allow Φ_{Flow} to know the type of data being visualized and choose an appropriate plot. For example, two spatial dimensions indicate a grid which might be plotted as a heatmap or vector field depending on the other dimensions. Batch dimensions translate to subfigures by default but can alternatively be used as the time axis in an animation. All plots shown in this document were created with one `plot()` call.

3. Experiments

We use the above features to solve two challenging inverse problems involving PDEs. Furthermore, we reimplement three experiments from prior work to show that Φ_{Flow} is broadly applicable.

Material composition from thermal conductivity First, we consider heat conduction involving 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 every point based on two snapshots of the temperature distribution. An exact reconstruction is impossible due to the limited information. Fig. 2 shows our implementation along with an example reconstruction. Us-



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

def simulate(x):  # x is the guess for the conductivity
    return diffuse.implicit(initial_temp, field.maximum(0, x), 10, Solve('biCG-stab(2)'))

boundary = {'x': 'periodic', 'y': 'zero-gradient'}
conductivity = field.maximum(0, CenteredGrid(Noise(scale=100), boundary, x=100, y=40))
initial_temp = CenteredGrid(Noise(), boundary, x=100, y=40)
final_temp = simulate(conductivity)

fit = minimize(lambda x: math.l2_loss(final_temp - simulate(x)),
               Solve('GD', x0=conductivity.with_values(.01)))
```

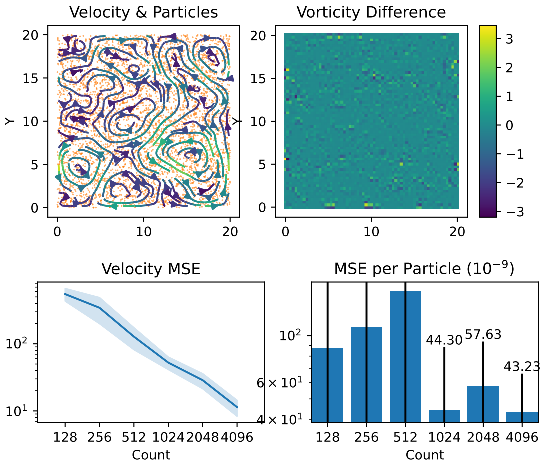
Figure 2. Conductivity reconstruction on a 100×40 grid with implicit heat diffusion. The conductivity distribution is optimized for using gradient descent with Jax, based on the initial and diffused state after 10 seconds.

ing Φ_{Flow}, we can easily write a differentiable simulator for implicit heat conduction, ensuring numerical stability. We use this to generate the ground-truth observation data and to optimize for the material composition by minimizing the MSE between the observed and reconstructed temperature profile with gradient descent. Differentiating w.r.t. the conductivity requires the implicit gradient w.r.t. the diffusion matrix. This would require manual gradient implementation in most ML libraries, but Φ_{Flow} supports matrix gradients for linear systems.

As can be seen in Fig. 2, fine detail cannot be recovered, but the fit approximates the ground truth conductivity distribution. The whole experiment can be written in 8 lines of code using Φ_{Flow}, attesting to the high information density. We encourage readers to read the source code, as we believe it explains our methodology in more detail and clarity than we can achieve in text.

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

$$v = \arg\min_{v'} ||M(M_0, \Delta t, v') - M_0||_2^2, \tag{1}$$



```python
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)),
                    Solve('L-BFGS-B', x0=0 * v0.downsample(4))).at(v0)
11  fit2 = minimize(lambda x: math.l2_loss(final_markers - simulate(x+fit1)),
                    Solve('L-BFGS-B', x0=0 * v0))
12  v_estimate = fit1 + fit2
```

Figure 3. Reconstruction of a divergence-free velocity field v from 128 to 2096 particles, observed at two times 0.1 seconds apart. We use L-BFGS-B to first fit v at quarter-resolution and afterwards for a residual fit at full resolution.

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. 3 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. 3, 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. 3 show that increasing the number of markers improves the velocity MSE at about one order of magnitude per 8× more markers. The average particle position MSE also decreases but is more variable across runs with large standard deviations.

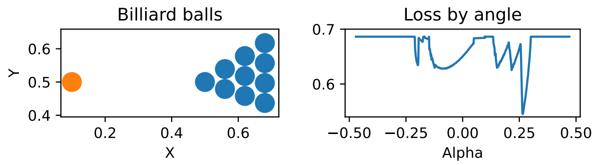
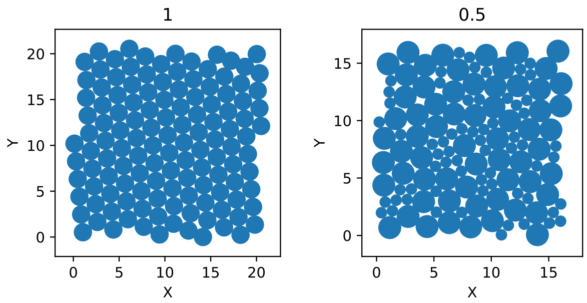


Figure 4. Replication of the billiards experiment from [21]. Setup with orange cue ball left, and loss $L(\alpha)$ right, where α denotes angle of the cue ball velocity v_{cue}. Source code in appendix C.



```
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
```

Figure 5. Replication of the sphere packing experiment from [20]. We use L-BFGS-B to find the maximally frustrated state.

Billiards This experiment [21] 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. 4. The task is finding the initial velocity of the cue ball v_{cue}, such that one corner ball from 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. 4.

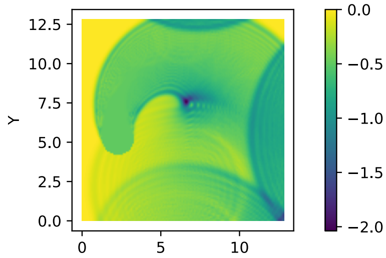


Figure 6. Replication of the wave propagation example from warp [11]. Source code in appendix C.

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.

Sphere packing The goal of this experiment [20] 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. 5 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 [11] consists of a wave simulation on a 128×128 grid, driven by a circular obstacle that moves inside the domain. Fig. 6 shows the final wave height after five seconds. Our implementation, shown in appendix C, consists of 14 lines of code, which is significantly shorter than the original warp implementation at 165 lines.

4. Conclusions

We have explained and 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 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 and appendix C demonstrates that user code written against Φ_{Flow}'s API is concise and expressive, without sacrificing flexibility or performance. To emphasize this point, we reimplemented three experiments that were published as showcases for their respective frameworks. In all cases, we achieve the same result with more concise, less convoluted code, making our implementation easier to understand. To assess the readability, we tasked ChatGPT with explaining our code and the original code, see Appendix C. 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.

References

[1] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia, “Learning to simulate complex physics with graph networks,” in *International Conference on Machine Learning*, PMLR, 2020, pp. 8459–8468.

[2] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, “Accelerating eulerian fluid simulation with convolutional networks,” in *International Conference on Machine Learning*, PMLR, 2017, pp. 3424–3433.

[3] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, “Machine learning–accelerated computational fluid dynamics,” *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, 2021.

[4] K. Um, R. Brand, Y. R. Fei, P. Holl, and N. Thuerey, “Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 6111–6122, 2020.

[5] K. Bieker, S. Peitz, S. L. Brunton, J. N. Kutz, and M. Dellnitz, “Deep model predictive flow control with limited sensor data and online learning,” *Theoretical and computational fluid dynamics*, vol. 34, no. 4, pp. 577–591, 2020.

[6] P. Holl, V. Koltun, and N. Thuerey, “Learning to control pdes with differentiable physics,” *arXiv preprint arXiv:2001.07457*, 2020.

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

[8] S. Greydanus, M. Dzamba, and J. Yosinski, “Hamiltonian neural networks,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[9] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2012, pp. 5026–5033.

[10] S. K. Mitusch, S. W. Funke, and J. S. Dokken, “Dolfin-adjoint 2018.1: Automated adjoints for fenics and firedrake,” *Journal of Open Source Software*, vol. 4, no. 38, p. 1292, 2019.

[11] M. Macklin, *Warp: A high-performance python framework for gpu simulation and graphics*, https://github.com/nvidia/warp, NVIDIA GPU Technology Conference (GTC), Mar. 2022.

[12] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[13] M. Abadi, P. Barham, J. Chen, *et al.*, “{Tensorflow}: A system for {large-scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[14] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.2.5, 2018. [Online]. Available: http://github.com/google/jax.

[15] I. Babuschkin, K. Baumli, A. Bell, *et al.*, *The DeepMind JAX Ecosystem*, 2020. [Online]. Available: http://github.com/deepmind.

[16] T. Hennigan, T. Cai, T. Norman, and I. Babuschkin, *Haiku: Sonnet for JAX*, version 0.0.3, 2020. [Online]. Available: http://github.com/deepmind/dm-haiku.

[17] S. Rasp and N. Thuerey, “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*, vol. 13, no. 2, e2020MS002405, 2021.

[18] S. Thapa, N. Li, and J. Ye, “Dynamic fluid surface reconstruction using deep neural network,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 21–30.

[19] K. Stachenfeld, D. B. Fielding, D. Kochkov, *et al.*, “Learned coarse models for efficient turbulence simulation,” *arXiv preprint arXiv:2112.15275*, 2021.

[20] S. Schoenholz and E. D. Cubuk, “Jax md: A framework for differentiable physics,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 428–11 441, 2020.

[21] Y. Hu, L. Anderson, T.-M. Li, *et al.*, “Difftaichi: Differentiable programming for physical simulation,” *arXiv preprint arXiv:1910.00935*, 2019.

[22] D. A. Bezgin, A. B. Buhendwa, and N. A. Adams, “Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows,” *Computer Physics Communications*, p. 108 527, 2022, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2022.108527. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465522002466.

[23] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, “Machine learning–accelerated computational fluid dynamics,” *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, 2021, ISSN: 0027-8424. DOI: 10.1073/pnas.2101784118. eprint: https://www.pnas.org/content/118/21/e2101784118.full.pdf. [Online]. Available: https://www.pnas.org/content/118/21/e2101784118.

[24] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[25] P. Holl, V. Koltun, and N. Thuerey, “Physical gradients for deep learning,” *arXiv preprint arXiv:2109.15048*, 2021.

[26] N. Thuerey, P. Holl, M. Mueller, P. Schnell, F. Trost, and K. Um, “Physics-based deep learning,” *arXiv preprint arXiv:2109.05237*, 2021.

[27] P. Schnell, P. Holl, and N. Thuerey, “Half-inverse gradients for physical deep learning,” *arXiv preprint arXiv:2203.10131*, 2022.

[28] P. Holl, V. Koltun, and N. Thuerey, “Scale-invariant learning by physics inversion,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 5390–5403, 2022.

[29] J. Brandstetter, D. E. Worrall, and M. Welling, “Message passing neural pde solvers,” in *International Conference on Learning Representations*, 2021.

[30] N. Wandel, M. Weinmann, and R. Klein, “Teaching the incompressible navier-stokes equations to fast neural surrogate models in 3d,” *arXiv preprint arXiv:2012.11893*, 2020.

[31] J. Brandstetter, R. van den Berg, M. Welling, and J. K. Gupta, "Clifford neural layers for pde modeling," in *The Eleventh International Conference on Learning Representations*, 2022.

[32] M. Takamoto, F. Alesiani, and M. Niepert, "Learning neural pde solvers with parameter-guided channel attention," *arXiv preprint arXiv:2304.14118*, 2023.

[33] O. Teikmanis, L. Leja, and K. Freivalds, "Applying a differentiable physics simulation to move objects with fluid streams,"

[34] N. Wandel, M. Weinmann, and R. Klein, "Teaching the incompressible navier–stokes equations to fast neural surrogate models in three dimensions," *Physics of Fluids*, vol. 33, no. 4, 2021.

[35] N. Wandel, M. Weinmann, and R. Klein, "Learning incompressible fluid dynamics from scratch-towards fast, differentiable fluid models that generalize," in *International Conference on Learning Representations*, 2020.

[36] G. Bokil, T. Geyer, and S. Wolff, "Towards convolutional neural networks for heat exchangers in electrified aircraft," 2023.

[37] V. Sengar, K. Seemakurthy, J. Gubbi, and B. P, "Multi-task learning based approach for surgical video desmoking," in *Proceedings of the twelfth Indian conference on computer vision, graphics and image processing*, 2021, pp. 1–9.

[38] N. Parekh, A. Zou, I. Jungling, K. Endlich, J. Sadowski, and M. Steinhausen, "Sex differences in control of renal outer medullary circulation in rats: Role of prostaglandins," *American Journal of Physiology-Renal Physiology*, vol. 264, no. 4, F629–F636, 1993.

[39] S. Brahmachary and N. Thuerey, "Unsteady cylinder wakes from arbitrary bodies with differentiable physics-assisted neural network," *arXiv preprint arXiv:2308.04296*, 2023.

[40] B. Ramos, F. Trost, and N. Thuerey, "Control of two-way coupled fluid systems with differentiable solvers," in *ICLR 2022 Workshop on Generalizable Policy Learning in Physical World*, 2022.

[41] V. C. Yadav *et al.*, "Simulation of fluid flows based on the data-driven evolution of vortex particles," Ph.D. dissertation, Institut für Strömungsmechanik, 2021.

[42] N. Tathawadekar, N. A. K. Doan, C. F. Silva, and N. Thuerey, "Hybrid neural network pde solvers for reacting flows," *arXiv preprint arXiv:2111.11185*, 2021.

[43] R. Wang, R. Walters, and R. Yu, "Approximately equivariant networks for imperfectly symmetric dynamics," in *International Conference on Machine Learning*, PMLR, 2022, pp. 23 078–23 091.

[44] J. A. Kurz, M. G. Seman, T. Khan, B. A. Bowman, and C. A. Oian, "Machine learning for pac1d and sese," 2023.

[45] R. Wang, R. Walters, and R. Yu, "Meta-learning dynamics forecasting using task inference," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 640–21 653, 2022.

[46] P. Wang, "The applications of generative adversarial network in surgical videos," in *Third International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI 2022)*, SPIE, vol. 12509, 2023, pp. 300–305.

[47] T. Wu, T. Maruyama, and J. Leskovec, "Learning to accelerate partial differential equations via latent global evolution," *Advances in Neural Information Processing Systems*, vol. 35, pp. 2240–2253, 2022.

[48] Z. Li, S. Patil, D. Shu, and A. B. Farimani, "Latent neural pde solver for time-dependent systems," in *NeurIPS 2023 AI for Science Workshop*, 2023.

[49] M. Takamoto, T. Praditia, R. Leiteritz, *et al.*, "Pdebench: An extensive benchmark for scientific machine learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 1596–1611, 2022.

[50] J. K. Gupta and J. Brandstetter, "Towards multi-spatiotemporal-scale generalized pde modeling," *arXiv preprint arXiv:2209.15616*, 2022.

[51] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[52] P. Holl and N. Thuerey, "Φ-ml: Intuitive scientific computing with dimension types for jax, pytorch, tensorflow & numpy," *Journal of Open Source Software*, vol. 9, no. 95, p. 6171, 2024. DOI: `10.21105/joss.06171`. [Online]. Available: `https://doi.org/10.21105/joss.06171`.

[53] T. C. et al., *Dlpack: Open in memory tensor structure*, `https://github.com/dmlc/dlpack`, 2017.

[54] J. H. Ferziger, M. Perić, and R. L. Street, *Computational methods for fluid dynamics*. springer, 2019.

[55] J. R. Shewchuk *et al.*, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[56] Y. Saad, "Iterative methods for sparse linear systems," *IEEE Computational Science and Engineering*, vol. 3, no. 4, p. 87, 1996.

[57] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: `10.1109/MCSE.2007.55`.

[58] P. T. Inc. "Collaborative data science." (2015), [Online]. Available: `https://plot.ly`.

[59] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: `http://github.com/google/jax`.

A. Additional Information on the Design and Implementation of $\Phi_{\textbf{Flow}}$

Simulation implementation For all simulations, users assemble a custom simulation function from the building blocks provided by Φ_{Flow}, such as diffusion, advection or linear system solves. We provide a large number of pre-built examples on our website, but always expose the simulation procedure directly to the user. This allows users to easily implement complex simulation schemes, from operator splitting approaches to the SIMPLE algorithm **patankar1980numerical**. Most of Φ_{Flow}'s functions can operate both on grids and meshes.

Spatial order of accuracy All differential operators can be configured in the spatial order to be used. For grids, all positive inegers are supported in theory, and all operators can be applied either explicitly or implicitly for added stability. Unstructured meshes support first and second-order as well as upwind schemes.

Solving linear systems of equations Φ_{Flow} includes all SciPy solvers, as well as custom GPU-compatible conjugate gradient and (stabilized) bi-conjugate gradient methods [55] 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 [56] and clustering, which can drastically improve convergence speed.

Linear solves are implemented differentiably, i.e. the adjoint system is solved during backpropagation or computation of higher-order derivatives. This is implemented via implicit differentiation in order to save on memory requirements and converge more quickly than backpropagating through the rolled-out solve. Users can specify the termination of the gradient solve and higher-order derivative solves independently of the forward solve if desired. Preconditioners are generally re-used in the adjoint solve.

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.

B. 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

C. 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 [59] 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 ($\Phi_{\textbf{Flow}}$) 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})

```python
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]), 0x00000, 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:

1. **Imports**: The code imports necessary libraries, including math for mathematical operations and a module named warp for parallel computation and rendering.
2. **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.
3. **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).
4. **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.
5. **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]
```

```python
    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()
```