JAX-SPH: A DIFFERENTIABLE SMOOTHED PARTICLE HYDRODYNAMICS FRAMEWORK

Artur P. Toshev[†,1], **Harish Ramachandran**[*,1], **Jonas A. Erbesdobler**[*,1], **Gianluca Galletti** [*,2] **Johannes Brandstetter**[3,4] **& Nikolaus A. Adams**[1,5]

[1] Chair of Aerodynamics and Fluid Mechanics, TUM, Germany
[2] Independent researcher
[3] ELLIS Unit Linz, LIT AI Lab, Institute for Machine Learning, JKU Linz, Austria
[4] NXAI GmbH, Linz, Austria
[5] Munich Institute of Integrated Materials, Energy and Process Engineering, TUM, Germany
[†] artur.toshev@tum.de
[*] Equal contribution

ABSTRACT

Particle-based fluid simulations have emerged as a powerful tool for solving the Navier-Stokes equations, especially in cases that include intricate physics and free surfaces. The recent addition of machine learning methods to the toolbox for solving such problems is pushing the boundary of the quality vs. speed trade-off of such numerical simulations. In this work, we lead the way to Lagrangian fluid simulators compatible with deep learning frameworks, and propose JAX-SPH – a Smoothed Particle Hydrodynamics (SPH) framework implemented in JAX. JAX-SPH builds on the code for dataset generation from the LagrangeBench project (Toshev et al., 2023b) and extends this code in multiple ways: (a) integration of further key SPH algorithms, (b) restructuring the code toward a Python library, (c) verification of the gradients through the solver, and (d) demonstration of the utility of the gradients for solving inverse problems as well as a Solver-in-the-Loop application. Our code is available at https://github.com/tumaer/jax-sph.

1 INTRODUCTION

Partial differential equations (PDEs) are the mathematical tools developed to describe natural phenomena ranging from engineering and physics to social sciences. Various numerical methods have been developed to solve these PDEs, as analytical solutions are only available for toy examples, with the most recent class of PDE solvers being the machine learning-based ones (Thuerey et al., 2021; Brunton & Kutz, 2023). One particular class of machine learning (ML) approaches called hybrid solvers refers to combining ideas (or full algorithmic blocks) from classical numerical solvers and machine learning (Schenck & Fox, 2018; Um et al., 2020; Kochkov et al., 2021; Jagtap et al., 2022; Lienen & Günnemann, 2022; Karlbauer et al., 2022; Li & Farimani, 2022; Toshev et al., 2023c).

This has been one of the main reasons for the development of differentiable fluid mechanics solvers like PhiFlow (Holl et al., 2020), JAX-CFD (Kochkov et al., 2021), and JAX-Fluids (Bezgin et al., 2022). However, these three frameworks implement Eulerian, i.e., grid-based, solvers, and we lead the way to a JAX-based Lagrangian CFD solver. Eulerian solvers refer to numerical methods that discretize space into static volume elements and then track the evolution of the fluid properties at these positions, while Lagrangian solvers discretize individual material elements, which are then shifted in space following the local velocity field.

Algorithmically, grid-based and particle-based methods are very different. While grid-based solvers rely on stencils akin to kernels in Convolutional Neural Networks (CNNs) (Lecun et al., 1998), particle-based solvers rely on kernel approximations akin to Graph Neural Networks (GNNs) (Scarselli et al., 2008; Battaglia et al., 2018) operating on a radial distance-based graph. The main overhead of Lagrangian over Eulerian approaches is updating the connectivity between discretization elements at every autoregressive solver step. Even if an Eulerian scheme operates on

an irregular mesh, the connectivity prescribed by this mesh can be precomputed, while by moving particles in space in Lagrangian methods, their neighbors constantly vary in time.

As SPH techniques advance, various software packages have arisen in the past few years. However, most of them are designed for high-performance computing (HPC) applications and are typically implemented in low-level languages like C++ or Fortran (Crespo et al., 2015; Koschier et al., 2019; Zhang et al., 2021), with two notable exceptions: PySPH (Ramachandran et al., 2021) in Python and juSPH (Luo et al., 2022) in Julia. Nevertheless, hardly an SPH implementation exists which is out-of-the-box compatible with modern deep learning frameworks like TensorFlow (Abadi et al., 2015), PyTorch[1] (Paszke et al., 2019) or JAX (Bradbury et al., 2018), i.e., leverages automatic differentiation to power differentiable solvers like PhiFlow (Holl et al., 2020). Upon identifying the lack of an ML-ready SPH solver, we choose JAX for the framework implementation for two reasons: (a) JAX tends to be faster for operations on graphs even after PyTorch 2.0 has been introduced (see appendix F in Toshev et al. (2023b)), and (b) we can use the cell list-based neighbor search routine of the JAX-MD library (Schoenholz & Cubuk, 2020). We note that with our solver, we target the easier integration of SPH with ML workflows rather than developing a parallel HPC code, and we exclusively use the AD routine `grad` by JAX. We leave the implementation of better custom adjoints along the lines of Ma et al. (2021); Kidger (2022); Nadarajah & Jameson (2000) to future work.

In this work, we significantly extend the codebase used for dataset generation within LagrangeBench[2] (Toshev et al., 2023b) and demonstrate the utility of the obtained gradients in multiple ways. Our contributions are:

- The addition and validation of Transport Velocity (Adami et al., 2013), Riemann SPH (Zhang et al., 2017b), and thermal diffusion effects (Cleary, 1998).
- Validating the accuracy of the automatic differentiation-based gradients (Griewank & Walther, 2008) over a few solver steps.
- A restructured code package, which we will open-source as a Python library.
- A demonstration of how to use these gradients on (a) an inverse problem over 100 SPH solver steps, and (b) using the SPH solver in a Solver-in-the-Loop fashion (Um et al., 2020).

2 SPH Solver

In this section, we introduce the components included in our solver code, and in appendix A, we include various solver validation cases.

Weakly compressible SPH. We follow the weakly compressible SPH approach (Monaghan, 1994; Morris et al., 1997) to evolve the dynamics of incompressible fluids. The equations governing such systems are the mass and momentum conservation equations

$$\frac{\mathrm{d}}{\mathrm{d}t}(\rho) = -\rho\left(\nabla\cdot\mathbf{u}\right), \tag{1}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{u}) = \underbrace{-\frac{1}{\rho}\nabla p}_{\text{pressure}} + \underbrace{\frac{1}{Re}\nabla^2\mathbf{u}}_{\text{viscosity}} + \underbrace{\mathbf{f}}_{\text{ext. force}}, \tag{2}$$

with density ρ, velocity $\mathbf{u}$, pressure p, Reynolds number Re, and external force $\mathbf{f}$. To numerically solve these equations, SPH applies a distance-based kernel W that averages over the properties of the fluid. The default kernel in our codebase is the Quintic spline (Morris et al., 1997), but we also include the 5th order Wendland kernel (Wendland, 1995).

The preferred way of estimating the density (Monaghan, 2005) is through *density summation* $\rho_i = \sum_j m_j W(r_{ij}|h)$, with m_j being the mass of the adjacent particles j, h the smoothing length of the kernel, and r_{ij} the interparticle distance. However, this approach leads to unphysically low

[1]For similar work in progress, we refer the reader to TorchSPH https://github.com/wi-re/pytorchSPH. Compared to our JAX-SPH, this repo uses PyTorch, different SPH algorithms, different experiments, and does not validate the solver or the gradients.

[2]https://github.com/tumaer/lagrangebench/blob/main/notebooks/data_gen.ipynb.

densities at free surfaces, forcing the use of *density evolution*, i.e., numerically integrating the mass conservation Eq. 1. We also implement a density field reinitialization method (Zhang et al., 2017a) to mitigate errors arising from density evolution (Colagrossi & Landrini, 2003). In weakly compressible SPH, the pressure is defined as a function of density through an equation of state, and in our code, we choose the formulation by Monaghan (1994).

Transport velocity. The observation that tensile instabilities in standard SPH can cause particle clumping and void regions (Price, 2012) has led to the development of advanced shifting schemes like the transport velocity formulation of SPH (Adami et al., 2013; Zhang et al., 2017a). We add the shifting velocity proposed by Adami et al. (2013) as an optional feature in our codebase.

Riemann SPH. We also add the Riemann SPH solver formulation by Zhang et al. (2017b), which is based on the idea of introducing a simple low dissipation limiter to a classical Riemann solver, resulting in decreased numerical dissipation. We construct an inter-particle Riemann problem along a unit vector pointing from particle i to particle j, which implicitly regularizes both the momentum equation and the mass conservation. We implement the dissipation limiter as proposed by Zhang et al. (2017b).

Wall boundaries. We follow the generalized wall boundary condition approach by Adami et al. (2012) to enforce different boundary conditions. This approach, in essence, implements two physical constraints: (a) impermeability of walls and (b) viscous effects at walls. For impermeability, the pressure of the adjacent fluid particles is assigned to the dummy wall particles, which leads to a zero pressure gradient in the wall-normal direction at the wall surface, and, thus prevents the penetration of fluid particles into the wall. Regarding viscosity, there are two cases to distinguish: (b1) *no-slip* enforces the no-slip boundary condition, i.e., the fluid must have zero velocity directly at the wall surface, and (b2) *free-slip*, i.e., the fluid must have a zero velocity normal to a wall, but might have any velocity tangentially. These two conditions are implemented by assigning the inverted velocity of the fluid to the wall particles, either fully for (b1) or only in the wall-normal direction for (b2). See Adami et al. (2012) for more details.

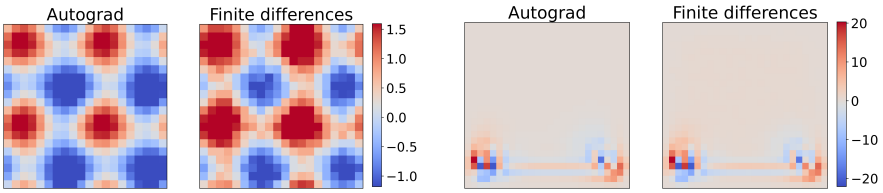
Thermal diffusion. Thermal diffusion in weakly compressible SPH involves the transfer of heat between neighboring particles governed by Fourier's law of heat (Cleary, 1998). This diffusion smoothens the temperature field by applying an SPH kernel interpolation over adjacent particles to compute the rate of temperature change. As we deal with incompressible fluids, the dynamics are not influenced by thermal effects, and the pressure or velocity fields do not directly influence the temperature field. Thus, the addition of temperature does not interfere with the previously presented SPH algorithms and allows us to study thermal effects which are governed by diffusion (by our explicit diffusion modeling) and convection (as Lagrangian particles are shifted in space). For an example of a channel flow with a hot wall, see appendix C.

3 EXPERIMENTS

Gradient validation. To test the validity of the gradients obtained through our differentiable solver, we compute analytical solver gradients via automatic differentiation and compare those to numerical gradients from finite difference schemes (Griewank & Walther, 2008). In our setup, gradients are accumulated over 5 solver steps, preceded by 10 (forward only) state warm-up steps. Epsilon for finite differences is picked as $0.001dx = 5e-5$, as smaller values lead to instabilities. Fig. 1 shows the scalar gradients of kinetic energy over position changes $\frac{dE_{kin}}{dr}$ when using 2-dimensional Taylor-Green vortex (TGV) (Brachet et al., 1983; 1984) and lid-driven cavity (LDC) (Ghia et al., 1982) as initial states.

Inverse problem. Our first application case is an inverse problem, representing the class of inverse design and flow control problems that are successfully tackled by differentiable solvers (Holl et al., 2020) as well as differentiable learned solver surrogates (Allen et al., 2022). The scenario involves a 2D box containing a water cube, discretized by 36 particles, which undergoes acceleration due to gravity across 100 solver steps. The task is to find the initial coordinates of the cube if the final coordinates are given; see Fig. 2.

The inverse problem is formulated by computing the mean squared error of coordinates between the target final state and the end of a simulation with randomly placed initial particles. After as few as



(a) TGV (b) LDC

Figure 1: Gradient magnitudes with JAX Autograd and with finite differences on Taylor-Green vortex (left) and lid-driven cavity (right).

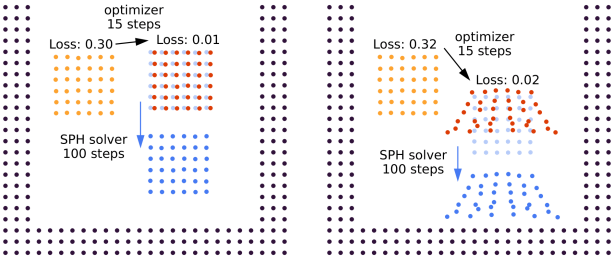


Figure 2: Inverse problems of finding the initial coordinates (light blue) given the final coordinates (blue) of a falling water cube simulation spanning 100 SPH steps. The optimization spans 15 gradient descent steps from orange to red. Free fall case (left) and wall-interactions (right).

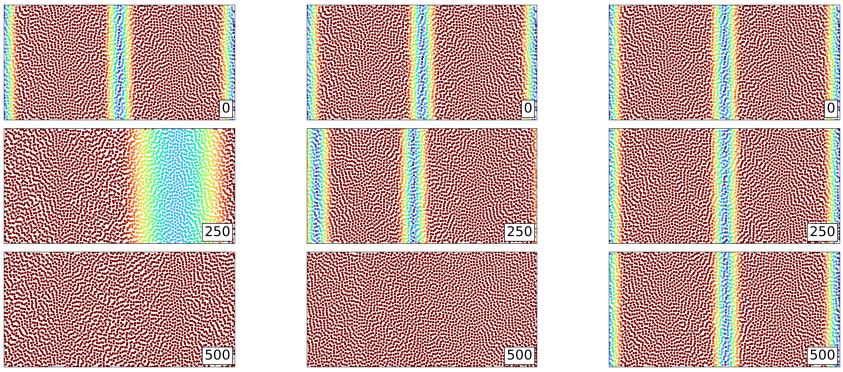
15 gradient descent steps, we reach a state closely resembling the original one, up to some loss of information during the deformation of the water cube while interacting with the wall.

Solver-in-the-loop. As a second experiment to showcase the solver differentiability, we adapt the popular *"Solver-in-the-Loop"* (SitL) (Um et al., 2020) training scheme to SPH. Initially developed to tackle spatial coarsening on grids, SitL interleaves a solver operating on a coarse spatial and/or temporal discretization with a learnable correction function. The solver manages coarse, low-frequency components, while the learnable function adjusts high-frequency details. Due to the inherent difficulties of spatial coarse-graining in particle systems, our objective is to implement SitL only for temporal coarsening. This significantly differs from the original application of SitL and mandates a series of design changes to the original architecture, mainly related to the normalization and training procedure, see appendix B.

Fig. 3 shows the time evolution of a 2D reverse Poiseuille flow (RPF) (Fedosov et al., 2008) dataset similar to the one in Toshev et al. (2023b) but consisting of positions sampled every 20th ground truth SPH step. Fig. 3 compares only employing the coarse SPH solver with $L = 3$ intermediate steps (left), a fully learned GNS model (Sanchez-Gonzalez et al., 2020) without intermediate steps (middle), and SitL using the same GNS model, but having three intermediate steps of GNS and SPH (right). More details on the training and quantitative results can be found in appendix B.2.

4 Conclusion

We have developed JAX-SPH, a framework for simulating Lagrangian fluid problems, which can be easily integrated into design/control problems as well as hybrid solver approaches. By building our code on the high-performance library JAX and validating the simulation results of our solver, we offer a fast and reliable SPH solver in Python. With our work, we hope to accelerate the development of more hybrid Lagrangian solvers, e.g., Toshev et al. (2024), and we leave the addition of more SPH algorithms and simulation cases to future work. One particularly exciting future direction is developing foundation models for PDEs that can operate on both Eulerian and Lagrangian data (Alkin et al., 2024), potentially combined with encoded symmetries (Toshev et al., 2023a).



(a) Solver only ($L = 3$) (b) Learned only ($L = 1$) (c) SitL ($L = 3$)

Figure 3: Evolution of the velocity magnitude in reverse Poiseuille flow. The stamps in each image refer to the respective step of the original SPH simulation, i.e., 500 means 10 000 SPH steps.

ACKNOWLEDGEMENTS

The authors thank Fabian Fritz for providing an initial JAX implementation of the 3D Taylor-Green vortex simulated with the transport velocity SPH formulation by Adami et al. (2013). The authors also thank Xiangyu Hu and Christopher Zöller for helpful discussions on SPH, and Ludger Paehler for discussions on gradient validation.

AUTHOR CONTRIBUTIONS

A.T. developed the codebase, selected the SPH algorithms and validated most of them, designed the simulation cases and experiments, ran many of them, and wrote the first version of the manuscript. H.R. implemented thermal diffusion and the inverse problem, and helped in the initial phase of Solver-in-the-Loop. J.E. implemented Riemann SPH, generated the final solver validation results, and contributed to refactoring the core SPH code. G.G. validated the gradients through the solver, implemented Solver-in-the-Loop, and tuned its parameters on the dataset A.T. provided. J.B. contributed to the choice of experiments showcasing the gradients through the solver. N.A. supervised the project from conception to design of experiments and analysis of the results. All authors contributed to the manuscript.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

Stefan Adami, Xiangyu Hu, and Nikolaus A Adams. A generalized wall boundary condition for smoothed particle hydrodynamics. *Journal of Computational Physics*, 231(21):7057–7075, 2012.

Stefan Adami, XY Hu, and Nikolaus A Adams. A transport-velocity formulation for smoothed particle hydrodynamics. *Journal of Computational Physics*, 241:292–307, 2013.

Benedikt Alkin, Andreas Fürst, Simon Schmid, Lukas Gruber, Markus Holzleitner, and Johannes Brandstetter. Universal physics transformers, 2024.

Kelsey R. Allen, Tatiana Lopez-Guevara, Kimberly L. Stachenfeld, Alvaro Sanchez-Gonzalez, Peter W. Battaglia, Jessica B. Hamrick, and Tobias Pfaff. Inverse design for fluid-structure interactions using graph network simulators, 2022.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Deniz A. Bezgin, Aaron B. Buhendwa, and Nikolaus A. Adams. 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.

Marc E Brachet, Daniel I Meiron, Steven A Orszag, BG Nickel, Rudolf H Morf, and Uriel Frisch. Small-scale structure of the taylor–green vortex. *Journal of Fluid Mechanics*, 130:411–452, 1983.

Marc E Brachet, D Meiron, S Orszag, B Nickel, R Morf, and Uriel Frisch. The taylor-green vortex and fully developed turbulence. *Journal of Statistical Physics*, 34(5-6):1049–1063, 1984.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

Steven L. Brunton and J. Nathan Kutz. Machine Learning for Partial Differential Equations. *arXiv preprint arXiv:2303.17078*, March 2023.

Paul W Cleary. Modelling confined multi-material heat and mass flows using sph. *Applied Mathematical Modelling*, 22(12):981–993, 1998.

Andrea Colagrossi and Maurizio Landrini. Numerical simulation of interfacial flows by smoothed particle hydrodynamics. *Journal of computational physics*, 191(2):448–475, 2003.

Alejandro JC Crespo, José M Domínguez, Benedict D Rogers, Moncho Gómez-Gesteira, S Longshaw, RJFB Canelas, Renato Vacondio, Anxo Barreiro, and Orlando García-Feal. Dualsphysics: Open-source parallel cfd solver based on smoothed particle hydrodynamics (sph). *Computer Physics Communications*, 187:204–216, 2015.

Dmitry A Fedosov, Bruce Caswell, and George Em Karniadakis. Reverse poiseuille flow: the numerical viscometer. In *Aip conference proceedings*, volume 1027, pp. 1432–1434. American Institute of Physics, 2008.

UKNG Ghia, Kirti N Ghia, and CT Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of computational physics*, 48(3):387–411, 1982.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Philipp Holl, Nils Thuerey, and Vladlen Koltun. Learning to control pdes with differentiable physics. In *International Conference on Learning Representations*, 2020.

Ameya D Jagtap, Zhiping Mao, Nikolaus Adams, and George Em Karniadakis. Physics-informed neural networks for inverse problems in supersonic flows. *Journal of Computational Physics*, 466:111402, 2022.

Matthias Karlbauer, Timothy Praditia, Sebastian Otte, Sergey Oladyshkin, Wolfgang Nowak, and Martin V Butz. Composing partial differential equations with physics-aware neural networks. In *International Conference on Machine Learning*, pp. 10773–10801. PMLR, 2022.

Patrick Kidger. On neural differential equations, 2022.

Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21):e2101784118, 2021.

Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids. In Wenzel Jakob and Enrico Puppo (eds.), *Eurographics 2019 - Tutorials*. The Eurographics Association, 2019. doi: 10.2312/egt.20191035.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Zijie Li and Amir Barati Farimani. Graph neural network-accelerated lagrangian fluid simulation. *Computers & Graphics*, 103:201–211, 2022.

Marten Lienen and Stephan Günnemann. Learning the dynamics of physical systems from sparse observations with finite element networks. In *International Conference on Learning Representations (ICLR)*, 2022.

Mimi Luo, Jiayu Qin, and Gang Mei. jusph: A julia-based open-source package of parallel smoothed particle hydrodynamics (sph) for dam break problems. *SoftwareX*, 19:101151, 2022.

Yingbo Ma, Vaibhav Dixit, Michael J Innes, Xingjian Guo, and Chris Rackauckas. A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9. IEEE, 2021.

Joe J Monaghan. Simulating free surface flows with sph. *Journal of computational physics*, 110(2): 399–406, 1994.

Joe J Monaghan. Smoothed particle hydrodynamics. *Reports on progress in physics*, 68(8):1703, 2005.

Joseph P Morris, Patrick J Fox, and Yi Zhu. Modeling low reynolds number incompressible flows using sph. *Journal of computational physics*, 136(1):214–226, 1997.

Siva Nadarajah and Antony Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In *38th Aerospace sciences meeting and exhibit*, pp. 667, 2000.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Daniel J Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *Journal of Computational Physics*, 231(3):759–794, 2012.

Prabhu Ramachandran, Aditya Bhosale, Kunal Puri, Pawan Negi, Abhinav Muta, A. Dinesh, Dileep Menon, Rahul Govind, Suraj Sanka, Amal S. Sebastian, Ananyo Sen, Rohan Kaushik, Anshuman Kumar, Vikas Kurapati, Mrinalgouda Patil, Deep Tavker, Pankaj Pandey, Chandrashekhar Kaushik, Arkopal Dutt, and Arpit Agarwal. Pysph: A python-based framework for smoothed particle hydrodynamics. *ACM Trans. Math. Softw.*, 47(4), sep 2021. ISSN 0098-3500. doi: 10.1145/3460773.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, 2020.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks, 2018.

Samuel S. Schoenholz and Ekin D. Cubuk. Jax m.d. a framework for differentiable physics. In *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. Physics-based Deep Learning. *arXiv preprint arXiv:2109.05237*, 2021.

Artur P. Toshev, Gianluca Galletti, Johannes Brandstetter, Stefan Adami, and Nikolaus A. Adams. Learning lagrangian fluid mechanics with $E(3)$-equivariant graph neural networks. In Frank Nielsen and Frédéric Barbaresco (eds.), *Geometric Science of Information*, pp. 332–341, Cham, 2023a. Springer Nature Switzerland. ISBN 978-3-031-38299-4.

Artur P. Toshev, Gianluca Galletti, Fabian Fritz, Stefan Adami, and Nikolaus A. Adams. Lagrangebench: A lagrangian fluid mechanics benchmarking suite, 2023b.

Artur P. Toshev, Ludger Paehler, Andrea Panizza, and Nikolaus A. Adams. On the relationships between graph neural networks for the simulation of physical systems and classical numerical methods, 2023c.

Artur P. Toshev, Jonas A. Erbesdobler, Nikolaus A. Adams, and Johannes Brandstetter. Neural sph: Improved neural modeling of lagrangian fluid dynamics, 2024.

Kiwon Um, Robert Brand, Yun Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers. *Advances in Neural Information Processing Systems*, 2020.

Holger Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in computational Mathematics*, 4:389–396, 1995.

Chi Zhang, Xiangyu Y Hu, and Nikolaus A Adams. A generalized transport-velocity formulation for smoothed particle hydrodynamics. *Journal of Computational Physics*, 337:216–232, 2017a.

Chi Zhang, XY Hu, and Nikolaus A Adams. A weakly compressible sph method based on a low-dissipation riemann solver. *Journal of Computational Physics*, 335:605–620, 2017b.

Chi Zhang, Massoud Rezavand, Yujie Zhu, Yongchuan Yu, Dong Wu, Wenbin Zhang, Jianhang Wang, and Xiangyu Hu. Sphinxsys: An open-source multi-physics and multi-resolution library based on smoothed particle hydrodynamics. *Computer Physics Communications*, 267:108066, 2021.

A SOLVER VALIDATION

A.1 TAYLOR GREEN VORTEX 2D

Fig. 4 shows the absolute velocities of each particle at the start and at the end of the Taylor Green vortex (Brachet et al., 1983; 1984) simulation at $Re = 100$. Here, the transport velocity formulation of SPH is used. In Fig. 5, one can see the decay of the maximum velocity and the kinetic energy over time for the different solvers, standard SPH (SPH), transport velocity formulation SPH (SPH + tvf), and Riemann SPH (Riemann).

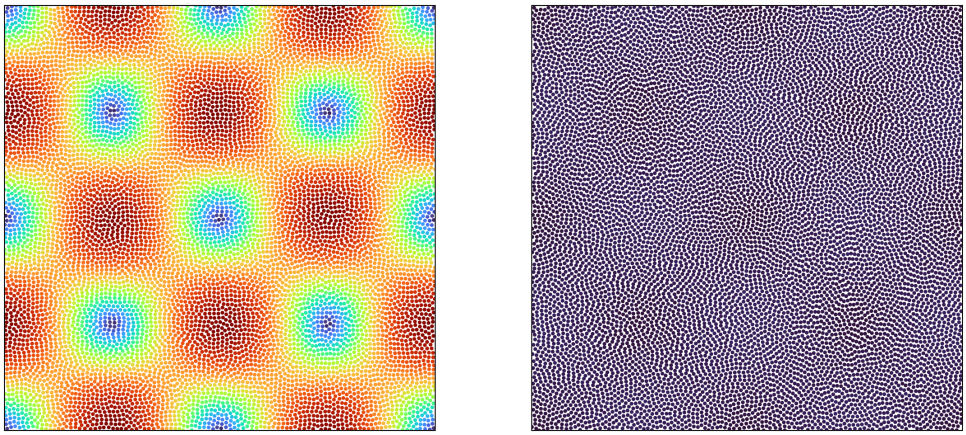


Figure 4: 2D Taylor Green vortex velocity magnitudes at the start of the simulation $t = 0$ (left) and at $t = 5$ (right), calculated using transport velocity formulation SPH.

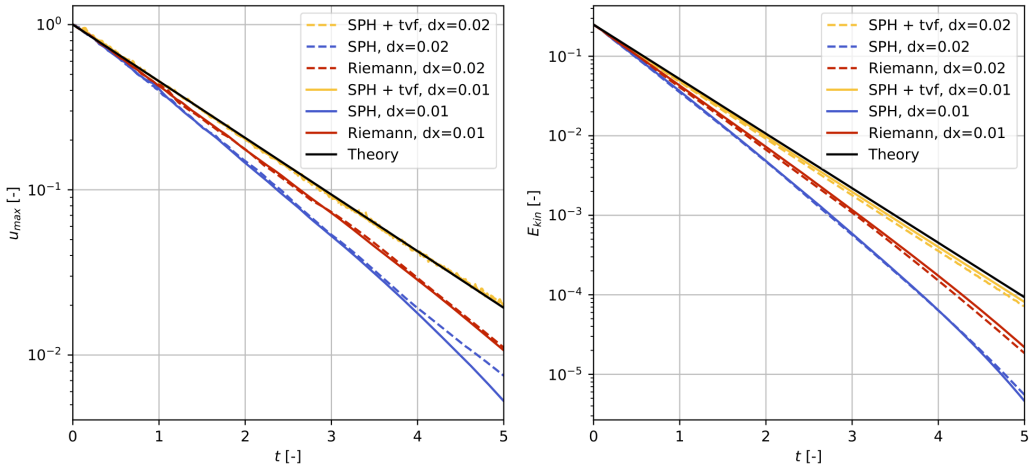


Figure 5: 2D Taylor Green vortex solver comparison for u_{max} (left) and E_{kin} (right) between standard SPH, transport velocity formulation SPH, and Riemann SPH at $dx = 0.02$ and $dx = 0.01$.

A.2 LID-DRIVEN CAVITY 2D

The following figures compare a 2D lid-driven cavity (Ghia et al., 1982) simulation for the different solvers at $Re = 100$. The reference data for the velocity profiles, i.e., black dots for U (velocity in x direction at a vertical cut through the middle of the cavity) and black squares for V (velocity in y direction at a horizontal cut through the middle of the cavity), are from Ghia et al. (1982).

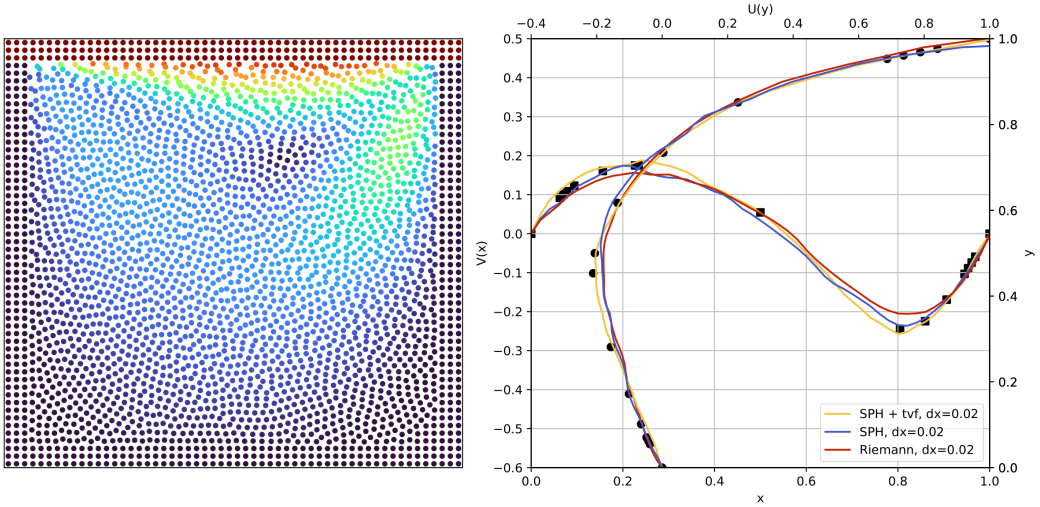


Figure 6: Lid-driven cavity with $dx = 0.02$ showing absolute particle velocities of the Riemann solver (left) and velocity profiles of each solver at the midsection for U and V (right)

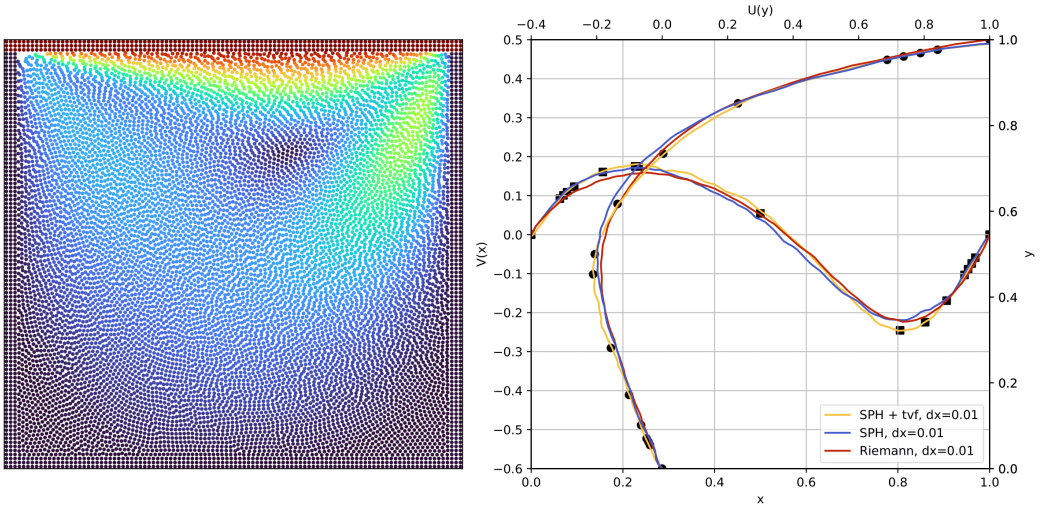


Figure 7: Lid-driven cavity with $dx = 0.01$ showing absolute particle velocities of the Riemann solver (left) and velocity profiles of each solver at the midsection for U and V (right)

A.3 Dam Break 2D

The following Fig. 8 shows the nondimensionalized pressure of a Riemann SPH (Zhang et al., 2017b) dam break (Colagrossi & Landrini, 2003) simulation at different time stamps. The fluid flows from the initial state on the left to the right side, interacts with the wall, and reflects a wave backward throughout the domain. A similar validation plot for the standard SPH formulation of our solver is presented in Toshev et al. (2023b).

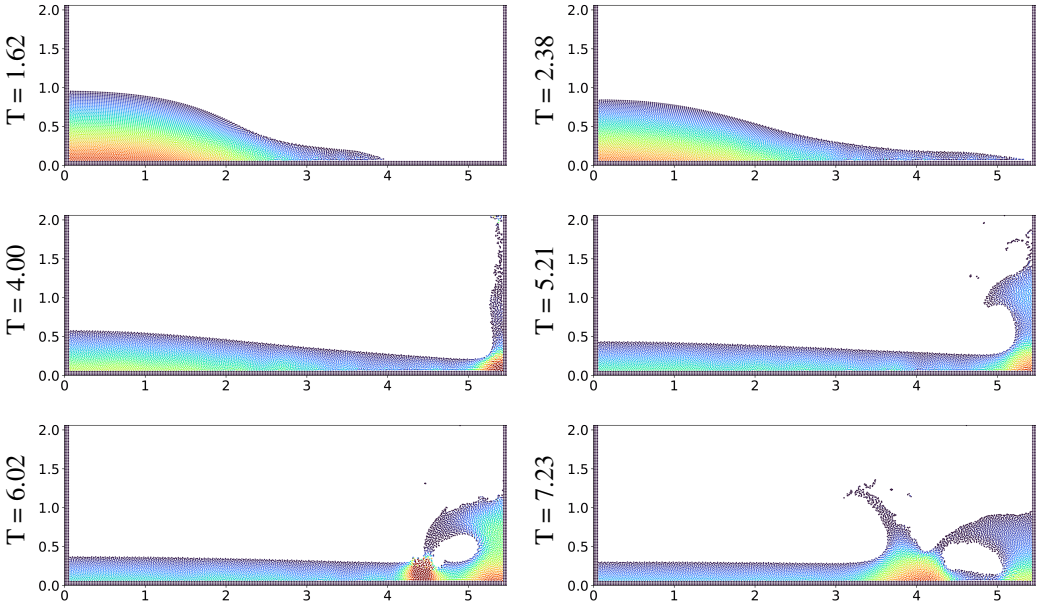


Figure 8: Dam break simulation with $dx = 0.02$ using Riemann SPH at different time stamps T, visualizing the nondimensionalized pressure

A.4 TAYLOR-GREEN VORTEX 3D

Fig. 9 compares the solvers on 3D TGV, similarly to Fig. 5 for 2D. Again, the Reynolds number is set to $Re = 100$, and the number of particles in the unit cube is 20^3, 32^3, and 50^3, leading to $dx = 0.314$, $dx = 0.196$, and $dx = 0.126$, respectively. The reference solution is obtained using JAX-Fluids Bezgin et al. (2022) with a 128^3 grid.

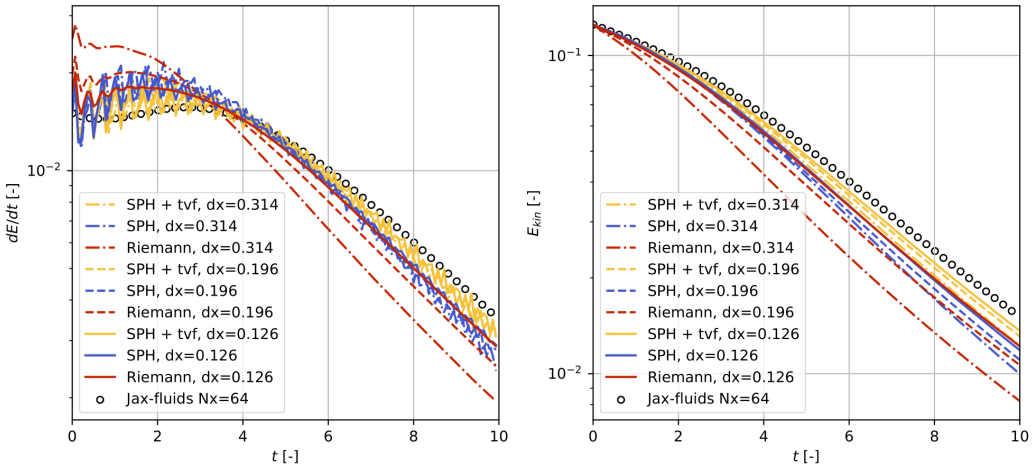


Figure 9: 3D Taylor Green vortex solver comparison for dE/dt (left) and E_{kin} (right) between standard SPH, transport velocity formulation SPH, and Riemann SPH with $dx = 0.314$, $dx = 0.196$, and $dx = 0.126$

B SOLVER-IN-THE-LOOP

B.1 IMPLEMENTATION DETAILS

To compute velocities, we apply a finite difference approximation of positions as

$$\mathbf{u}_{(i+1)M} = \mathbf{x}_{(i+1)M} - \mathbf{x}_{iM}, \tag{3}$$

$$\mathbf{a}_{iM} = \mathbf{u}_{(i+1)M} - \mathbf{u}_{iM}. \tag{4}$$

These properties have normalization statistics over the dataset as σ_{uM} and σ_{aM} (for brevity here, assuming that everything is zero-centered, but we center the data in our code). The values entering and exiting the SitL model should be in normalized space.

Given the number of temporal coarsening steps M and the number of SitL calls `n_sitl`, each call counts for $\frac{M}{\texttt{n_sitl}}$ actual timesteps. This results in two distinct time steps (dt), necessitating the transformation of input properties to the SitL physical space.

```python
def sitl_forward(self, r, u_M_norm, nbrs_GNN, dt, M, n_sitl, sigma_uM, sigma_aM):
    """Solver-in-the-loop forward call.

    Args:
        r: current coordinates
        u_M_norm: normalized position difference between M SPH steps.
        nbrs_GNN: GNN neighbor list
        dt: physical dt from CFL condition
        M: level of temporal coarsening
        n_sitl: number of Solver-in-the-Loop steps
        sigma_uM: std of u_M over the dataset (= std_dx_M)
        sigma_aM: std of a_M over the dataset (= std_ddx_M)
    """
    # transform initial states
    u_phys = u_M_norm * sigma_uM / (dt * M)  # to physical
    nbrs_SPH = self.nbrs_SPH_update(r)  # internal neighbor list
    r0, u0, r_new = r.copy(), u_phys.copy(), r

    for l in range(n_sitl):
        # SPH solver call
        a_SPH = SPH(u_phys, nbrs_SPH)

        # learned correction
        a_GNN = GNN(u_M_norm, nbrs_GNN)
        a_GNN_phys = a_GNN * sigma_aM / (dt * M) ** 2  # to physical

        # add accelerations (in physical space) and integrate
        a_final = a_SPH + a_GNN_phys
        u_phys += (dt * M) / n_sitl * a_final
        r_new += (dt * M) / n_sitl * u_phys

        # update neighbors to new positions
        nbrs_SPH = self.nbrs_SPH_update(r_new)
        nbrs_GNN = nbrs_GNN_update(r_new)

        # normalize updated velocity for next GNN input
        u_M_norm = u_phys * (dt * M) / sigma_uM  # to normalized

    # finite differences to get M-step effective quantities
    dx_M = r_new - r0
    dx_M_norm = dx_M / sigma_uM
    ddx_M = dx_M - u0 * (dt * M)
    ddx_M_norm = ddx_M / sigma_aM

    return {"acc": ddx_M_norm, "vel": dx_M_norm}
```

Listing 1: Solver-in-the-Loop forward algorithm

B.2 TRAINING AND RESULTS

Solver-in-the-Loop was trained with LagrangeBench (Toshev et al., 2023b), with a custom RPF 2D dataset with 20-step temporal coarsening. For SitL, `n_sitl`=3. Both the corrector model and the reference GNS (Sanchez-Gonzalez et al., 2020) are message-passing networks with 10 layers and 64 latent dimensions. The starting learning rate is set to $1e-3$ for both, and `noise_std` is set to $1e-5$ for SitL and to $1e-3$ for GNS. Table 1 shows the LagrangeBench performance metrics on these models. Best models are picked based on the MSE_{20} loss on the validation set.

Metric	Solver only	GNS	SitL
MSE_5	$1.7e-7$	$6.7e-9$	$\mathbf{3.3e-9}$
MSE_{20}	$7.9e-6$	$1.9e-7$	$\mathbf{1.3e-7}$
$\text{MSE}_{E_{kin}}$	0.13	$2.8e-4$	$\mathbf{7.4e-5}$
Sinkhorn	$3.4e-7$	$3.7e-8$	$\mathbf{9.3e-9}$

Table 1: LagrangeBench metrics on the RPF 2D dataset over 20 steps.

C THERMAL DIFFUSION EXAMPLE

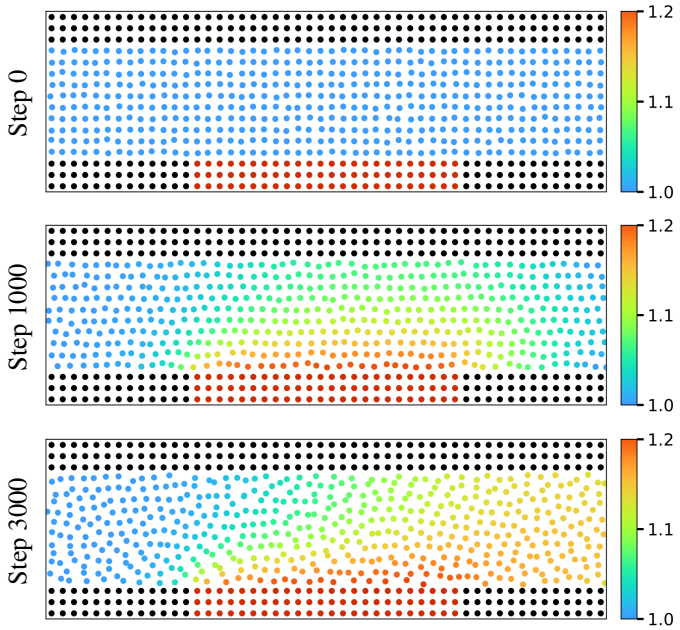


Figure 10: Simulation of channel flow with hot bottom wall using standard SPH and thermal diffusion. The plots show the non-dimensional temperature at different time steps of an SPH simulation.