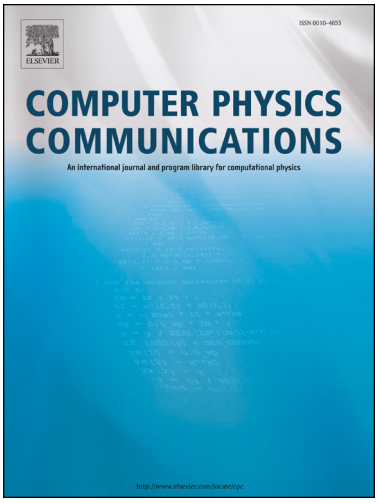
Journal Pre-proof



JAX-Fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows

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

PII: S0010-4655(22)00246-6
DOI: https://doi.org/10.1016/j.cpc.2022.108527
Reference: COMPHY 108527

To appear in: *Computer Physics Communications*

Received date: 20 April 2022
Revised date: 24 August 2022
Accepted date: 1 September 2022

Please cite this article as: 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*, 108527, doi: https://doi.org/10.1016/j.cpc.2022.108527.

JAX-Fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows

Deniz A. Bezgin[1,*], Aaron B. Buhendwa[1,*], Nikolaus A. Adams

Technical University of Munich, School of Engineering and Design, Chair of Aerodynamics and Fluid Mechanics, Boltzmannstr. 15, 85748 Garching bei München, Germany

Abstract

Physical systems are governed by partial differential equations (PDEs). The Navier-Stokes equations describe fluid flows and are representative of nonlinear physical systems with complex spatio-temporal interactions. Fluid flows are omnipresent in nature and engineering applications, and their accurate simulation is essential for providing insights into these processes. While PDEs are typically solved with numerical methods, the recent success of machine learning (ML) has shown that ML methods can provide novel avenues of finding solutions to PDEs. ML is becoming more and more present in computational fluid dynamics (CFD). However, up to this date, there does not exist a general-purpose ML-CFD package which provides 1) powerful state-of-the-art numerical methods, 2) seamless hybridization of ML with CFD, and 3) automatic differentiation (AD) capabilities. AD in particular is essential to ML-CFD research as it provides gradient information and enables optimization of preexisting and novel CFD models. In this work, we propose JAX-Fluids: a comprehensive fully-differentiable CFD Python solver for compressible two-phase flows. JAX-Fluids is intended for ML-supported CFD research. The framework allows the simulation of complex fluid dynamics with phenomena like three-dimensional turbulence, compressibility effects, and two-phase flows. Written entirely in JAX, it is straightforward to include existing ML models into the proposed framework. Furthermore, JAX-Fluids enables end-to-end optimization. I.e., ML models can be optimized with gradients that are backpropagated through the entire CFD algorithm, and therefore contain not only information of the underlying PDE but also of the applied numerical methods. We believe that a Python package like JAX-Fluids is crucial to facilitate research at the intersection of ML and CFD and may pave the way for an era of differentiable fluid dynamics.

Keywords: Computational fluid dynamics, Machine learning, Differential programming, Navier-Stokes equations, Level-set, Turbulence, Two-phase flows

PROGRAM SUMMARY

*Corresponding author

Email addresses: deniz.bezgin@tum.de (Deniz A. Bezgin), aaron.buhendwa@tum.de (Aaron B. Buhendwa), nikolaus.adams@tum.de (Nikolaus A. Adams)

[1]Both authors contributed equally

Program Title: JAX-Fluids

CPC Library link to program files: `https://doi.org/10.17632/pzvkwn5s6p.1`

Developer's repository link: (`https://github.com/tumaer/JAXFLUIDS`)

Code Ocean capsule: `https://codeocean.com/capsule/6819679`

Licensing provisions: GNU GPLv3

Programming language: Python

Supplementary material: Source code; Examples; Videos: Moving solid bodies, Taylor-Green vortex, Rising bubble, Shock-bubble interaction.

Nature of problem: The compressible Navier-Stokes equations describe continuum-scale fluid flows. These flows often involve highly complex flow phenomena such as shocks, material interfaces, and turbulence. The intrinsic nonlinear dynamics render the numerical simulation of the these equations challenging. Machine learning provides novel avenues for describing partial differential equations. Machine learning models rely on gradient information provided by automatic differentiation and are often implemented in Python. In contrast, existing high-performance computational fluid dynamics codes are typically written in Fortran or C++ and do not offer inherent automatic differentiation capabilities. These discrepancies hinder the advance of machine-learning-supported computational fluid dynamics. Up to this day, a general-purpose fully-differentiable computational fluid dynamics solver for compressible two-phase flows is missing.

Solution method: We introduce JAX-Fluids: a general-purpose three-dimensional fully-differentiable computational fluid dynamics solver for compressible two-phase flows. JAX-Fluids is a simulation framework intended for machine-learning-supported computational fluid dynamics research. Our framework is written entirely in JAX, a high-performance numerical computing library with automatic differentiation capabilities. We have used an object-oriented programming style and a modular design philosophy. This allows the straightforward exchange of numerical methods. We provide a wide variety of state-of-the-art high-order computational methods for compressible flows. The modularity of our framework additionally facilitates the integration of custom subroutines. We use the sharp-interface level-set method to model two-phase flows. The software package can easily be installed as a Python package. We have build the source code around the JAX NumPy API. This makes JAX-Fluids accessible and performant. JAX-Fluids runs on CPUs, GPUs, and TPUs. We use HDF5 in combination with XDMF for writing output quantities. The Python packages Haiku and Optax are used for implementation and training of machine learning methods.

Additional comments including restrictions and unusual features: JAX-Fluids relies on open-source third-party Python libraries. These are automatically installed. In the current version, JAX-Fluids only runs on a single accelerator (CPU/GPU/TPU). Future versions will include support for parallel execution. JAX-Fluids has been tested on Linux and macOS operating systems.

1. Introduction

The evolution of most known physical systems can be described by partial differential equations (PDEs). Navier-Stokes equations (NSE) are partial differential equations that describe the continuum-

scale flow of fluids. Fluid flows are omnipresent in engineering applications and in nature, and the accurate numerical simulation of complex flows is crucial for the prediction of global weather phenomena [1, 2], for applications in biomedical engineering such as air flow through the lungs or blood circulation [3, 4], and for the efficient design of turbomachinery [5], wind turbines [6, 7], and airplane wings [8]. Computational fluid dynamics (CFD) aims to solve these problems with numerical algorithms.

While classical CFD has a rich history, in recent years the symbiosis of machine learning (ML) and CFD has sparked a great interest amongst researchers [9, 10, 11]. The amalgamation of classical CFD and ML requires powerful novel algorithms which allow seamless integration of data-driven models and, more importantly, end-to-end automatic differentiation (AD) [12]. Here, we provide JAX-Fluids (`https://github.com/tumaer/JAXFLUIDS`): the first state-of-the-art fully-differentiable CFD framework for the computation of three-dimensional compressible two-phase flows with high-order numerical methods. Over the course of this paper, we discuss the challenges of hybrid ML-accelerated CFD solvers and highlight how novel architectures like JAX-Fluids have the potential to facilitate ML-supported fluid dynamics research.

The quest for powerful numerical fluid dynamics algorithms has been a long-lasting challenge. In the mid of the last century, the development of performant computer processing units (CPUs) laid the fundament for the development of computational fluid dynamics. For the first time, computers were used to simulate fluid flows [13]. Computational fluid dynamics, i.e., the numerical investigation of fluid flows, became a scientific field on its own. With the rapid development of computational hardware, the CFD community witnessed new and powerful algorithms for the computation of more and more complex flows. Among others, robust time integration schemes, high-order spatial discretizations, and accurate flux-functions were thoroughly investigated. In the 1980s and 1990s, many advancements in numerical methods for compressible fluid flows followed, e.g., [14, 15, 16, 17, 18].

In recent years, machine learning has invigorated the physical sciences by providing novel tools for predicting the evolution of physical systems. Breakthrough inventions in ML [19, 20] and rapid technical developments of graphics processing units (GPUs) have led to an ever-growing interest in machine learning. Starting from applications in computer vision [21, 22], cognitive sciences [23], and genomics [24], machine learning and data-driven methods have also become more and more popular in physical and engineering sciences. This development has partially been fuelled by the emergence of powerful general-purpose automatic differentiation frameworks, such as Tensorflow [25], PyTorch [26], and JAX [27]. Natural and engineering sciences can profit from ML methods as they have capabilities to learn complex relations from data and enable novel strategies for modelling physics as well as new avenues of post-processing. For example, machine learning has been successfully used to identify PDEs from data [28], and physics-informed neural networks provide new ways for solving inverse problems [29, 30] by combining data and PDE knowledge.

It is well-known that fluid dynamics is a data-rich and compute-intensive discipline [11]. The nonlinearity of the underlying governing equations, the Navier-Stokes equations for viscous flows and the Euler equations for inviscid flows, is responsible for very complex spatio-temporal features of fluid flows. For example, turbulent flows exhibit chaotic behavior with strong intermittent flow features and non-Gaussian statistics. At the same time, in the inviscid Euler equations strong discontinuities can form over time due to the hyperbolic character of the underlying PDEs.

Machine learning offers novel data-driven methods to tackle long-standing problems in fluid dynamics [9, 11, 31]. A multitude of research has put forward different ways of incorporating ML in CFD applications. Applications range from fully data-driven surrogate models to less invasive hybrid data-driven numerical methods. Thereby, scientific ML methods can be categorized according to different criteria.

One important distinction is the level of physical prior-knowledge that is included in model and training [32]. Entirely data-driven ML models have the advantage of being quickly implemented and efficient during inference. However, they typically do not offer guarantees on performance (e.g., convergence, stability, and generalization), convergence in training is challenging, and it is often difficult to enforce physical constraints such as symmetries or conservation of energy. In contrast, established numerical methods are consistent and enforce physical constraints. Recently, intense research has investigated the hybridization of ML and classical numerical methods.

A second major distinction of ML models can be made according to on- and offline training. Up until now, ML models have been typically optimized offline, i.e., outside of physics simulators. Upon proper training, they are then plugged into an existing CFD solver for evaluation of down-stream tasks. Examples include training of explicit subgrid scale models in large eddy simulations [33], interface reconstruction in multiphase flows [34, 35], and cell face reconstruction in shock-capturing schemes [36, 37].

Although the offline training of ML models is relatively easy, there are several drawbacks to this approach. For one, these models suffer from a data-distribution mismatch between the data seen at training and test time. Secondly, they generally do not directly profit from a priori knowledge about the dynamics of the underlying PDE. Additionally, fluid mechanics solvers are often very complex, written in low-level programming languages like Fortran or C++, and heavily optimized for CPU computations. This is in contrast with practices in ML research: ML models are typically trained in Python and optimized for GPU usage. Inserting these models into preexisting CFD software frameworks can be a tedious task.

To tackle this problem, researchers have come up with differentiable CFD frameworks written entirely in Python which allow end-to-end optimization of ML models. The end-to-end (online) training approach utilizes automatic differentiation [12] through entire simulation trajectories. ML models trained in such a fashion experience the PDE dynamics and also see their own outputs during training.

While AD has become popular in recent years, adjoint operators promote an alternative to obtain gradients and have long been used in CFD [38]. In recent works, neural networks have been trained using a combination of AD and adjoint equations [39, 40]. Albeit solving adjoint equations is computationally effective, they require algorithmic and implementational overhead. Adjoint equations have to be derived and implemented for each quantity of interest - especially, a new adjoint equation has to be derived when the objective function changes. An automatically-differentiable code framework - like JAX-Fluids - only requires the user to implement the forward pass, the backward pass (i.e., gradient computation) is tracked automatically. This process is less error-prone and more flexible, facilitating reduced development times. Making use of AD, Bar-Sinai and co-workers [41, 42] have proposed a differentiable framework for finding optimal discretizations for simple non-linear one-dimensional problems and turbulent mixing in two-dimensions. Owoyele et al. [43] presented a neural ordinary differential equation approach to predict chemical source terms in combustion systems. In [44], a differentiable solver was placed in the training loop to reduce the error of iterative solvers. Bezgin et al. [45] have put forward a subgrid scale model for nonclassical shocks. Kochkov et al. [46] have optimized a subgrid scale model for two-dimensional turbulence.

Afore-noted works focus on simpler flow configurations. The problems are often one- and two-dimensional, incompressible, and lack complex physics such as two-phase flows, three-dimensional turbulence, or compressibility effects. Additionally, the written code packages often are highly problem specific and cannot be used as general differentiable CFD software packages. To the knowledge of the authors, despite a high-interest in ML-accelerated CFD, to this day there does not exist a comprehensive mesh-based software package for *differentiable fluid dynamics* in the Eulerian reference frame. At the same time, for Lagrangian frameworks, the differentiable software package JAX-MD [47] has been successfully used for molecular dynamics and provides a general fundament for other particle-based discretization methods.

We reiterate that the steady rise and success of machine learning in computational fluid dynamics, but also more broadly in computational physics, calls for a new generation of algorithms which allow

1. rapid prototyping in high-level programming languages,
2. algorithms which can be run on CPUs, GPUs, and TPUs,
3. the seamless integration of machine learning models into solver frameworks,
4. fully-differentiable algorithms which allow end-to-end optimization of data-driven models.

Realizing the advent of *differentiable fluid dynamics* and the increasing need for differentiable general high-order CFD solvers, here we introduce JAX-Fluids as a fully-differentiable general-purpose 3D finite-volume CFD solver for compressible two-phase flows. JAX-Fluids is written entirely in JAX [27], a numerical computing library with automatic differentiation capabilities which has seen increased popularity in the physical science community over the past several years. JAX-Fluids provides a wide variety of state-of-the-art high-order methods for compressible turbulent flows. A powerful level-

set implementation allows the simulation of arbitrary solid boundaries and two-phase flows. High-order shock-capturing methods enable the accurate computation of shock-dominated compressible flow problems and shock-turbulence interactions.

Performance and usability were key design goals during development of JAX-Fluids. The solver can conveniently be installed as a Python package. The source code builds on the JAX NumPy API. Therefore, JAX-Fluids is accessible, performant, and runs on accelerators like CPUs, GPUs, and TPUs. Additionally, an object-oriented programming style and a modular design philosophy allows users the easy integration of new modules. We believe that frameworks like JAX-Fluids are crucial to facilitate research at the intersection of ML and CFD, and may pave the way for an era of *differentiable fluid dynamics*.

The remainder of this paper is organized as follows. In Sections 2 and 3, we describe the physical and numerical model. Section 4 describes challenges of writing a high-performance CFD code in a high-level programming language such as JAX. We additionally detail the general structure of our research code. In Section 5, we evaluate our code as a classical physics simulator. We validate the numerical framework on canonical test cases from compressible fluid mechanics, including strong-shocks and turbulent flows. In Section 6, we assess the computational performance of JAX-Fluids. Section 7 showcases how the proposed framework can be used in machine learning tasks. Specifically, we demonstrate full-differentiability of the framework by optimizing a numerical flux function. Finally, in Section 8 we conclude and summarize the main results of our work.

2. Physical Model

We are interested in the compressible Euler equations for inviscid flows and in the compressible Navier-Stokes equations (NSE) which govern viscous flows. The state of a fluid at any position in the flow field $\mathbf{x} = [x, y, z]^T = [x_1, x_2, x_3]^T$ at time t can be described by the vector of primitive variables $\mathbf{W} = [\rho, u, v, w, p]^T$. Here, ρ is the density, $\mathbf{u} = [u, v, w]^T = [u_1, u_2, u_3]^T$ is the velocity vector, and p is the pressure. An alternative description is given by the vector of conservative variables $\mathbf{U} = [\rho, \rho u, \rho v, \rho w, E]^T$. Here, $\rho\mathbf{u} = [\rho u, \rho v, \rho w]^T$ are the momenta in the three spatial dimensions and $E = \rho e + \frac{1}{2}\rho\mathbf{u} \cdot \mathbf{u}$ is the total energy per unit volume. e is the internal energy per unit mass.

In differential formulation, the compressible Euler equations can be written in terms of $\mathbf{U}$

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathcal{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathcal{G}(\mathbf{U})}{\partial y} + \frac{\partial \mathcal{H}(\mathbf{U})}{\partial z} = 0. \tag{1}$$

The convective physical fluxes $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$ are defined as

$$\mathcal{F}(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E+p) \end{pmatrix}, \quad \mathcal{G}(\mathbf{U}) = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ v(E+p) \end{pmatrix}, \quad \mathcal{H}(\mathbf{U}) = \begin{pmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ w(E+p) \end{pmatrix}. \tag{2}$$

This set of equations must be closed by an equation of state (EOS) which relates pressure with density and internal energy, i.e., $p = p(\rho, e)$. Unless specified otherwise, we use the stiffened gas equation

$$p(\rho, e) = (\gamma - 1)\rho e - \gamma B, \tag{3}$$

where γ represents the ratio of specific heats and B is the background pressure.

The compressible Navier-Stokes equations can be seen as the viscous extension of the Euler equations. As before, we write them in terms of the conservative state vector $\mathbf{U}$,

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathcal{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathcal{G}(\mathbf{U})}{\partial y} + \frac{\partial \mathcal{H}(\mathbf{U})}{\partial z} = \frac{\partial \mathcal{F}^d(\mathbf{U})}{\partial x} + \frac{\partial \mathcal{G}^d(\mathbf{U})}{\partial y} + \frac{\partial \mathcal{H}^d(\mathbf{U})}{\partial z} + S(\mathbf{U}). \tag{4}$$

Here, we have introduced the dissipative fluxes $\mathcal{F}^d$, $\mathcal{G}^d$, and $\mathcal{H}^d$ and the source term vector $S(\mathbf{U})$ on the right-hand side. The dissipative fluxes describe viscous effects and heat conduction, and are given by

$$\mathcal{F}^d(\mathbf{U}) = \begin{pmatrix} 0 \\ \tau^{11} \\ \tau^{12} \\ \tau^{13} \\ \sum_i u_i \tau^{1i} - q_1 \end{pmatrix}, \quad \mathcal{G}^d(\mathbf{U}) = \begin{pmatrix} 0 \\ \tau^{21} \\ \tau^{22} \\ \tau^{23} \\ \sum_i u_i \tau^{2i} - q_2 \end{pmatrix}, \quad \mathcal{H}^d(\mathbf{U}) = \begin{pmatrix} 0 \\ \tau^{31} \\ \tau^{32} \\ \tau^{33} \\ \sum_i u_i \tau^{3i} - q_3 \end{pmatrix}. \tag{5}$$

The stresses τ^{ij} are given by

$$\tau^{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \delta_{ij} \frac{\partial u_k}{\partial x_k}. \tag{6}$$

μ is the dynamic viscosity. The energy flux vector $\mathbf{q}$ can be expressed via Fourier's heat conduction law, $\mathbf{q} = [q_1, q_2, q_3]^T = -\lambda \nabla T$. λ is the heat conductivity. The source terms $S(\mathbf{U})$ represent body forces or heat sources. The body force resulting from gravitational acceleration $\mathbf{g} = [g_1, g_2, g_3]^T$ is given by

$$S(\mathbf{U}) = \begin{pmatrix} 0 \\ \rho \mathbf{g} \\ \rho \mathbf{u} \cdot \mathbf{g} \end{pmatrix}. \tag{7}$$

In Table 1, we summarize the nomenclature and the reference values with which we non-dimensionalize aforementioned equations.

Quantitiy	Nomenclature	Reference quantity
Density	ρ	ρ_{ref}
Length	(x, y, z) or (x_1, x_2, x_3)	l_{ref}
Velocity	(u, v, w) or (u_1, u_2, u_3)	u_{ref}
Temperature	T	T_{ref}
Time	t	$t_{ref} = l_{ref}/u_{ref}$
Pressure	p	$p_{ref} = \rho_{ref} u_{ref}^2$
Viscosity	μ	$\mu_{ref} = \rho_{ref} u_{ref} l_{ref}$
Surface tension coefficient	σ	$\sigma_{ref} = \rho_{ref} u_{ref}^2 l_{ref}$
Thermal conductivity	λ	$\lambda_{ref} = \rho_{ref} u_{ref}^3 l_{ref}/T_{ref}$
Gravitation	g	$g_{ref} = u_{ref}^2/l_{ref}$
Specific gas constant	$\mathcal{R}$	$\mathcal{R}_{ref} = u_{ref}^2/T_{ref}$
Mass	m	$m_{ref} = \rho_{ref} l_{ref}^3$
Mass flow	$\dot{m}$	$\dot{m}_{ref} = m_{ref}/t_{ref} = \rho_{ref} u_{ref} l_{ref}^2$

Table 1: Overview on nomenclature and nondimensionalization.

3. Numerical Model

In this section we detail the numerical methods of JAX-Fluids. Table 2 provides an overview on the implemented numerical methods.

3.1. Finite-Volume Discretization

The differential form of the conservation law in Equations (1) and (4) assume smooth solutions for which partial derivatives exist. In practice, we solve the integral form of the partial differential equations using the finite-volume method. We use cuboid cells on a Cartesian grid. In general, cell (i, j, k) has spatial extension Δx, Δy, and Δz in the spatial dimensions x, y, and z, respectively. We denote the corresponding cell volume as $V = \Delta x \Delta y \Delta z$. Often, we use cubic cells for which $\Delta x = \Delta y = \Delta z$.

In the finite-volume formulation we are interested in the spatio-temporal distribution of cell-averaged values which are defined by

$$\bar{\mathbf{U}}_{i,j,k} = \frac{1}{V} \int_{x_{i-\frac{1}{2},j,k}}^{x_{i+\frac{1}{2},j,k}} \int_{y_{i,j-\frac{1}{2},k}}^{y_{i,j+\frac{1}{2},k}} \int_{z_{i,j,k-\frac{1}{2}}}^{z_{i,j,k+\frac{1}{2}}} \mathbf{U} \mathrm{d}x \mathrm{d}y \mathrm{d}z. \tag{8}$$

After application of volume integration to Equation (4), the temporal evolution of the cell-averaged

value in cell (i, j, k) is given by

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\bar{\mathbf{U}}_{i,j,k} = &-\frac{1}{\Delta x}\left(\mathbf{F}_{i+\frac{1}{2},j,k} - \mathbf{F}_{i-\frac{1}{2},j,k}\right)\\ &-\frac{1}{\Delta y}\left(\mathbf{G}_{i,j+\frac{1}{2},k} - \mathbf{G}_{i,j-\frac{1}{2},k}\right)\\ &-\frac{1}{\Delta z}\left(\mathbf{H}_{i,j,k+\frac{1}{2}} - \mathbf{H}_{i,j,k-\frac{1}{2}}\right)\\ &+\frac{1}{\Delta x}\left(\mathbf{F}^d_{i+\frac{1}{2},j,k} - \mathbf{F}^d_{i-\frac{1}{2},j,k}\right)\\ &+\frac{1}{\Delta y}\left(\mathbf{G}^d_{i,j+\frac{1}{2},k} - \mathbf{G}^d_{i,j-\frac{1}{2},k}\right)\\ &+\frac{1}{\Delta z}\left(\mathbf{H}^d_{i,j,k+\frac{1}{2}} - \mathbf{H}^d_{i,j,k-\frac{1}{2}}\right)\\ &+\bar{\mathbf{S}}_{i,j,k}.\end{aligned} \tag{9}$$

$\mathbf{F}$, $\mathbf{G}$, and $\mathbf{H}$ are the convective intercell numerical fluxes across the cell faces in x-, y-, and z-direction, and $\mathbf{F}^d$, $\mathbf{G}^d$, and $\mathbf{H}^d$ are the numerical approximations to the dissipative fluxes in x-, y-, and z-direction. $\bar{\mathbf{S}}_{i,j,k}$ is the cell-averaged source term. Note that the convective and dissipative fluxes in Equation (9) are cell face averaged quantities. We use a simple one-point Gaussian quadrature to evaluate the necessary cell face integrals, however approximations of higher order are also possible, see for example [48]. For the calculation of the convective intercell numerical fluxes, we use WENO-type high-order discretization schemes [49] in combination with approximate Riemann solvers. Subsection 3.3 provides more details. Dissipative fluxes are calculated by central finite-differences, see Subsection 3.4 for details. Multidimensional settings are treated via the dimension-by-dimension technique in a straightforward manner, i.e., aforementioned steps are repeated for each spatial dimension separately.

3.2. Time Integration

The finite-volume discretization yields a set of ordinary differential equations (ODEs) (9) that can be integrated in time by an ODE solver of choice. We typically use explicit total-variation diminishing (TVD) Runge-Kutta methods [50]. The time step size is given by the CFL criterion as described in [51].

3.3. Convective Flux Calculation

For the calculation of the convective fluxes an approximative solution to a Riemann problem has to be found at each cell face. As we make use of the dimension-by-dimension technique, without loss of generality, we restrict ourselves to a one-dimensional setting in this subsection. We are interested in finding the cell face flux $\mathbf{F}_{i+\frac{1}{2}}$ at the cell face $x_{i+\frac{1}{2}}$.

We distinguish between two different methods for the calculation of the convective intercell fluxes: the *High-order Godunov* approach and the *Flux-splitting* approach. In the *High-order Godunov* approach, we first reconstruct flow states left and right of a cell face, and subsequently enter an approximate Riemann solver for the calculation of the convective intercell flux. In the *Flux-splitting* approach,

we first perform cell-wise flux-splitting, reconstruct left and right going fluxes for each cell face, and subsequently assemble the final flux. In the following, we briefly sketch both methods.

High-order Godunov approach

1. Apply WENO reconstruction on the primitive variable $\mathbf{W}_i$/the conservative variable $\mathbf{U}_i$ to obtain the cell face quantities $\mathbf{W}^{\pm}_{i+\frac{1}{2}}/\mathbf{U}^{\pm}_{i+\frac{1}{2}}$. Note that the reconstruction can be done directly in physical space or via transformation in characteristic space. High-order reconstruction in physical space can lead to spurious oscillations due to the interaction of discontinuities in different fields [52].
2. Transform the reconstructed primitive/conservative variables at the cell face into conservative/primitive cell face quantities: $\mathbf{W}^{\pm}_{i+\frac{1}{2}} \rightarrow \mathbf{U}^{\pm}_{i+\frac{1}{2}}$ / $\mathbf{U}^{\pm}_{i+\frac{1}{2}} \rightarrow \mathbf{W}^{\pm}_{i+\frac{1}{2}}$.
3. Compute the final flux with an appropriate flux function/approximate Riemann solver, e.g., HLL [53] or HLLC [17]:
$$\mathbf{F}_{i+\frac{1}{2}} = \mathbf{F}_{i+\frac{1}{2}}\left(\mathbf{U}^{-}_{i+\frac{1}{2}}, \mathbf{U}^{+}_{i+\frac{1}{2}}, \mathbf{W}^{-}_{i+\frac{1}{2}}, \mathbf{W}^{+}_{i+\frac{1}{2}}\right) \tag{10}$$

Flux-Splitting approach

1. At the cell face $x_{i+\frac{1}{2}}$ compute an appropriate average state from neighboring cells $\mathbf{U}_{i+\frac{1}{2}} = \mathbf{U}_{i+\frac{1}{2}}(\mathbf{U}_i, \mathbf{U}_{i+1})$ (e.g., by arithmetic mean or Roe average [54]) and the corresponding Jacobian $\mathbf{A}_{i+\frac{1}{2}} = \mathbf{A}_{i+\frac{1}{2}}\left(\mathbf{U}_{i+\frac{1}{2}}\right)$.
2. Eigenvalue decomposition of the Jacobian: $\mathbf{A}_{i+\frac{1}{2}} = \mathbf{R}_{i+\frac{1}{2}}\mathbf{\Lambda}_{i+\frac{1}{2}}\mathbf{R}^{-1}_{i+\frac{1}{2}}$, with the matrix of right eigenvectors $\mathbf{R}_{i+\frac{1}{2}}$, the matrix of left eigenvectors $\mathbf{R}^{-1}_{i+\frac{1}{2}}$, and the matrix of eigenvalues $\mathbf{\Lambda}_{i+\frac{1}{2}}$.
3. Transform the cell state $\mathbf{U}_i$ and the flux $\mathbf{F}_i$ to characteristic space: $\mathbf{V}_j = \mathbf{R}^{-1}_{i+\frac{1}{2}}\mathbf{U}_i$, $\mathbf{G}_j = \mathbf{R}^{-1}_{i+\frac{1}{2}}\mathbf{F}_i$.
4. Perform the user-specified flux splitting: $\hat{\mathbf{G}}^{\pm}_i = \frac{1}{2}\left(\mathbf{G}_i \pm \bar{\mathbf{\Lambda}}_{i+\frac{1}{2}}\mathbf{V}_i\right)$, where $\bar{\mathbf{\Lambda}}_{i+\frac{1}{2}}$ is the eigenvalue matrix of the respective flux-splitting scheme.
5. Apply WENO reconstruction on $\hat{\mathbf{G}}^{\pm}_i$ to obtain $\hat{\mathbf{G}}^{\pm}_{i+\frac{1}{2}}$ at the cell face $x_{i+\frac{1}{2}}$.
6. Assemble the final flux in characteristic space: $\hat{\mathbf{G}}_{i+\frac{1}{2}} = \hat{\mathbf{G}}^{+}_{i+\frac{1}{2}} + \hat{\mathbf{G}}^{-}_{i+\frac{1}{2}}$.
7. Transform the final flux back to physical space: $\mathbf{F}_{i+\frac{1}{2}} = \mathbf{R}_{i+\frac{1}{2}}\hat{\mathbf{G}}_{i+\frac{1}{2}}$.

3.4. Dissipative Flux Calculation

For the calculation of the dissipative fluxes, we have to evaluate derivatives at the cell faces. We do this using central finite-differences. As we do all computations in a Cartesian framework, central finite-differences can be evaluated directly at a cell face if the direction of the derivative is parallel to the cell face normal. For example, at the cell face $x_{i+1/2,j,k}$ the x-derivative of any quantity ψ is directly approximated with second-order or fourth-order central finite-differences,

$$\left.\frac{\partial\psi}{\partial x}\right|^{C2}_{x_{i+1/2,j,k}} = \frac{-\psi_{i,j,k} + \psi_{i+1,j,k}}{\Delta x}, \quad \left.\frac{\partial\psi}{\partial x}\right|^{C4}_{x_{i+1/2,j,k}} = \frac{\psi_{i-1,j,k} - 27\psi_{i,j,k} + 27\psi_{i+1,j,k} - \psi_{i+2,j,k}}{24\Delta x}. \tag{11}$$

If the cell face normal is perpendicular to the direction of the derivative, we use a two-step process to approximate the derivative. We first evaluate the derivative at the cell-centers and then use a central interpolation to obtain the value at the cell face of interest. Again, we use second-order or fourth-order order central finite-differences for the derivatives. Let us consider the y-derivative of the quantity ψ at the cell face $x_{i+\frac{1}{2},j,k}$ (the z-derivative is completely analogous). We first calculate the y-derivative at the cell centers

$$\left.\frac{\partial\psi}{\partial y}\right|^{C2}_{x_{i,j,k}} = \frac{-\psi_{i,j-1,k}+\psi_{i,j+1,k}}{\Delta y}, \quad \left.\frac{\partial\psi}{\partial y}\right|^{C4}_{x_{i,j,k}} = \frac{\psi_{i,j-2,k}-8\psi_{i,j-1,k}+8\psi_{i,j+1,k}-\psi_{i,j+2,k}}{12\Delta y}. \tag{12}$$

We subsequently interpolate these values to the cell face in x-direction,

$$\begin{aligned}
\left.\frac{\partial\psi}{\partial y}\right|_{x_{i+1/2,j,k}} &= \frac{1}{2}\left(\left.\frac{\partial\psi}{\partial y}\right|_{x_{i-1,j,k}} + \left.\frac{\partial\psi}{\partial y}\right|_{x_{i+1,j,k}}\right), \\
\left.\frac{\partial\psi}{\partial y}\right|_{x_{i+1/2,j,k}} &= \frac{1}{16}\left(-\left.\frac{\partial\psi}{\partial y}\right|_{x_{i-2,j,k}} + 9\left.\frac{\partial\psi}{\partial y}\right|_{x_{i-1,j,k}} + 9\left.\frac{\partial\psi}{\partial y}\right|_{x_{i+1,j,k}} - \left.\frac{\partial\psi}{\partial y}\right|_{x_{i+2,j,k}}\right).
\end{aligned} \tag{13}$$

3.5. Source Terms and Forcings

The source terms $S(\mathbf{U})$ represent body forces and heat sources. We use them to impose physical constraints, e.g., fixed mass flow rates or temperature profiles. These forcings are required to simulate a variety of test cases. Examples include channel flows or driven homogenous isotropic turbulence. Fixed mass flow rates are enforced with a PID controller minimizing the error between target and current mass flow rate $e(t) = \frac{\dot{m}_{\text{target}}-\dot{m}(t)}{\dot{m}_{\text{target}}}$. Here, the control variable is an acceleration $a_{\dot{m}}$ that drives the fluid in the prescribed direction. We denote the unit vector pointing towards the prescribed direction as $\mathbf{N}$. The controller variable and resulting source terms read

$$a_{\dot{m}} = K_p e(t) + K_I \int_0^t e(\tau)\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \qquad S(\mathbf{U}) = \begin{pmatrix} 0 \\ \rho a_{\dot{m}}\mathbf{N} \\ \rho a_{\dot{m}}\mathbf{u}\cdot\mathbf{N} \end{pmatrix}, \tag{14}$$

where K_p, K_I, and K_d are the controller parameters. The integral and derivative in Equation (14) are approximated with first order schemes. Fixed temperature profiles are enforced with a heat source $\dot{\omega}_T$. The heat source and resulting source term is given by

$$\dot{\omega}_T = \rho R \frac{\gamma}{\gamma-1}\frac{T_{\text{target}}-T}{\Delta t}, \qquad S(\mathbf{U}) = \dot{\omega}_T[0,0,0,0,1]^T. \tag{15}$$

3.6. Level-set Method for Two-phase Flows

We use the level-set method [55] to model two-phase flows with fluid-fluid and fluid-solid interfaces. In particular, we implement the sharp-interface level-set method proposed by Hu et al. [56], which is also used in the solver ALPACA [57, 51]. The interface is tracked by a scalar field ϕ whose values represent the signed distance from the interface of each cell center within the mesh of the finite-volume discretization. This implies that there is a positive phase ($\phi > 0$) and a negative phase ($\phi < 0$) with

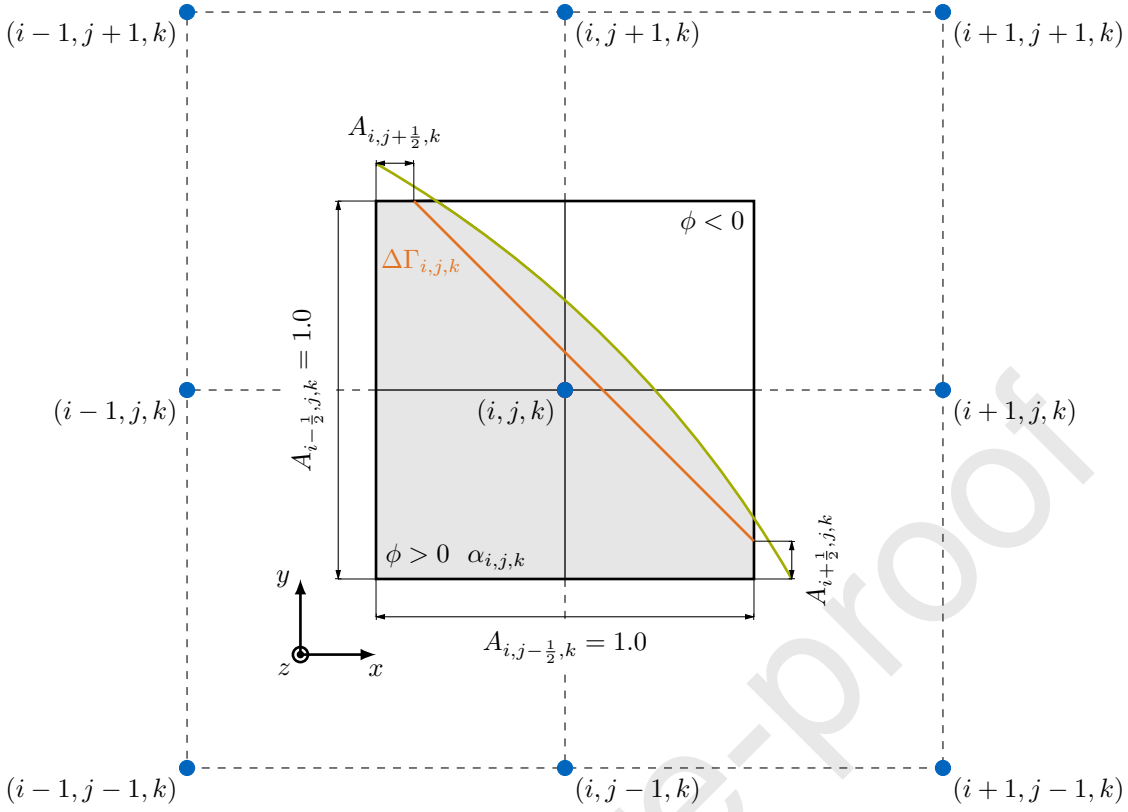


Figure 1: Schematic finite-volume discretization for cut cell (i, j, k) on a Cartesian grid. The red dots represent the cell centers. The red line indicates the interface, and the blue line gives the linear approximation of the interface. The fluid with positive level-set values is colored in gray, and the fluid with negative level-set values is colored in white. Volume fraction and apertures are computed for the positive fluid. Note that the figure illustrates a two-dimensional slice in the (x, y)-plane. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the interface being located at the zero level-set of ϕ. A cell that is intersected by the interface is referred to as cut cell. Figure 1 shows a schematic of a cut cell in the finite-volume discretization. The apertures $A_{i\pm\frac{1}{2},j,k}$, $A_{i,j\pm\frac{1}{2},k}$, and $A_{i,j,k\pm\frac{1}{2}}$ represent the portion of the cell face area that is covered by the respective fluid. The volume fraction $\alpha_{i,j,k}$ denotes the portion of the cell volume covered by the respective fluid. Hereinafter, we will refer to the positive phase with the subscript 1 and to the negative phase with the subscript 2. The following relations between the geometrical quantities for the positive and negative phase apply:

$$\alpha_1 = 1 - \alpha_2, \quad A_1 = 1 - A_2. \tag{16}$$

We solve Equation (9) for both phases separately. However, in a cut cell (i, j, k), the equation is

modified as follows.

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\alpha_{i,j,k}\bar{\mathbf{U}}_{i,j,k} = &-\frac{1}{\Delta x}\left[A_{i+\frac{1}{2},j,k}\left(\mathbf{F}_{i+\frac{1}{2},j,k}+\mathbf{F}^d_{i+\frac{1}{2},j,k}\right)-A_{i-\frac{1}{2},j,k}\left(\mathbf{F}_{i-\frac{1}{2},j,k}+\mathbf{F}^d_{i-\frac{1}{2},j,k}\right)\right]\\ &-\frac{1}{\Delta y}\left[A_{i,j+\frac{1}{2},k}\left(\mathbf{G}_{i,j+\frac{1}{2},k}+\mathbf{G}^d_{i,j+\frac{1}{2},k}\right)-A_{i,j-\frac{1}{2},k}\left(\mathbf{G}_{i,j-\frac{1}{2},k}+\mathbf{G}^d_{i,j-\frac{1}{2},k}\right)\right]\\ &-\frac{1}{\Delta z}\left[A_{i,j,k+\frac{1}{2}}\left(\mathbf{H}_{i,j,k+\frac{1}{2}}+\mathbf{H}^d_{i,j,k+\frac{1}{2}}\right)-A_{i,j,k-\frac{1}{2}}\left(\mathbf{H}_{i,j,k-\frac{1}{2}}+\mathbf{H}^d_{i,j,k-\frac{1}{2}}\right)\right]\\ &+\alpha_{i,j,k}\bar{\mathbf{S}}_{i,j,k}\\ &-\frac{1}{\Delta x\Delta y\Delta z}\left[\mathbf{X}_{i,j,k}(\Delta\Gamma)+\mathbf{X}^d_{i,j,k}(\Delta\Gamma)\right]\end{aligned} \tag{17}$$

The cell-averaged state and the intercell fluxes must be weighted with the volume fraction and the cell face apertures, respectively. The terms $\mathbf{X}(\Delta\Gamma)$ and $\mathbf{X}^d(\Delta\Gamma)$ denote the convective and dissipative interface fluxes, with $\Delta\Gamma$ being the interface segment length. We define the projections of the interface segment length on the x-, y-, and z-direction as the vector

$$\Delta\mathbf{\Gamma}_p = \begin{pmatrix}\Delta\Gamma(\mathbf{i}\cdot\mathbf{n}_I)\\ \Delta\Gamma(\mathbf{j}\cdot\mathbf{n}_I)\\ \Delta\Gamma(\mathbf{k}\cdot\mathbf{n}_I)\end{pmatrix}, \tag{18}$$

where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ represent the unit vectors in x-, y-, and z-direction, respectively. The interface normal is given by $\mathbf{n}_I = \nabla\phi/|\nabla\phi|$. The interface fluxes read

$$\mathbf{X} = \begin{pmatrix}0\\ p_I\Delta\mathbf{\Gamma}_p\\ p_I\Delta\mathbf{\Gamma}_p\cdot\mathbf{u}_I\end{pmatrix}, \qquad \mathbf{X}^d = \begin{pmatrix}0\\ \tau_I^T\Delta\mathbf{\Gamma}_p\\ (\tau_I^T\Delta\mathbf{\Gamma}_p)\cdot\mathbf{u}_I-\mathbf{q}_I\cdot\Delta\mathbf{\Gamma}_p\end{pmatrix}. \tag{19}$$

Here, p_I and $\mathbf{u}_I$ denote the interface pressure and interface velocity. The viscous interface stress tensor τ_I is given by

$$\tau_I = \begin{pmatrix}\tau_I^{11} & \tau_I^{12} & \tau_I^{13}\\ \tau_I^{21} & \tau_I^{22} & \tau_I^{23}\\ \tau_I^{31} & \tau_I^{32} & \tau_I^{33}\end{pmatrix}, \qquad \tau_I^{ij} = \mu_I\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)-\frac{2}{3}\mu_I\delta_{ij}\frac{\partial u_k}{\partial x_k}, \qquad \mu_I = \frac{\mu_1\mu_2}{\alpha_1\mu_2+\alpha_2\mu_1}. \tag{20}$$

The interface heat flux $\mathbf{q}_I$ reads

$$\mathbf{q}_I = -\lambda_I\nabla T, \qquad \lambda_I = \frac{\lambda_1\lambda_2}{\alpha_1\lambda_2+\alpha_2\lambda_1}. \tag{21}$$

The evaluation of τ_I and $\mathbf{q}_I$ requires the computation of velocity and temperature gradients at the interface. The gradient at the interface is approximated with the gradient at the cell center. We use the real fluid state to evaluate these gradients. The real fluid state in a cut cell is approximated by $\mathbf{W} = \alpha_1\mathbf{W}_1 + \alpha_2\mathbf{W}_2$. As we solve Equations (17) and (18) for each phase separately, we emphasize that $\mathbf{n}_I$, α, and A must be computed with respect to the present phase. Conservation is therefore satisfied since $\mathbf{n}_{I1} = -\mathbf{n}_{I2}$, i.e., the interface flux terms for two fluids at an interface have the same magnitude but opposite sign. The computation of the interface velocity $\mathbf{u}_I$ and interface pressure p_I depends on the type of interface interaction:

- For **fluid-solid** interactions, the interface velocity is prescribed as either a constant or a space- and time-dependent function. The interface pressure p_I is approximated with the cell pressure.
- For **fluid-fluid** interactions, the two-material Riemann problem at the interface is solved. The solution reads [58]

$$\begin{aligned} \mathbf{u}_I &= \frac{\rho_1 c_1 \mathbf{u}_1 \cdot \mathbf{n}_{I1} + \rho_2 c_2 \mathbf{u}_2 \cdot \mathbf{n}_{I1} + p_2 - p_1 - \sigma\kappa}{\rho_1 c_1 + \rho_2 c_2} \mathbf{n}_{I1}, \\ p_{I1} &= \frac{\rho_1 c_1 (p_2 + \sigma\kappa) + \rho_2 c_2 p_1 + \rho_1 c_1 \rho_2 c_2 (\mathbf{u}_2 \cdot \mathbf{n}_{I1} - \mathbf{u}_1 \cdot \mathbf{n}_{I1})}{\rho_1 c_1 + \rho_2 c_2}, \\ p_{I2} &= \frac{\rho_1 c_1 p_2 + \rho_2 c_2 (p_1 - \sigma\kappa) + \rho_1 c_1 \rho_2 c_2 (\mathbf{u}_2 \cdot \mathbf{n}_{I1} - \mathbf{u}_1 \cdot \mathbf{n}_{I1})}{\rho_1 c_1 + \rho_2 c_2}, \end{aligned} \tag{22}$$

  where c denotes the speed of sound, σ is the surface tension coefficient, and $\kappa = \nabla \cdot \mathbf{n}_{I1}$ denotes the curvature. For $\sigma \neq 0$ and $\kappa \neq 0$, the interface pressure experiences a jump at the interface as constituted by mechanical equilibrium. The interface pressure in Equation (19) must be chosen with respect to the present phase.

Assuming a linear interface within each cut cell, the cell face apertures are computed analytically as follows. The level-set values at the edges of a computational cell are computed using trilinear interpolation. The sign of the level-set values at the four corners of a cell face determine the cut cell face configuration. Figure 2 illustrates three typical cases of a cell face that is intersected by the interface. In total, there are 2^4 different sign combinations of the level-set values along the corners. Hence, there are 2^4 different cut cell face configurations. For each of these, the cell face aperture is evaluated as the (sum of) areas of the basic geometric shapes, i.e., triangle or trapezoid. The interface

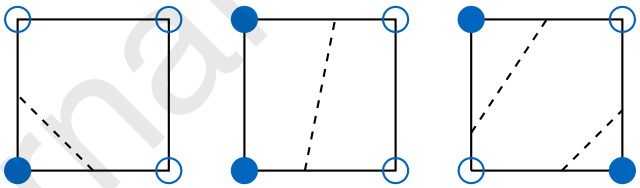


Figure 2: Three typical cut cell face configurations. Solid and hollow blue circles indicate positive and negative level-set corner values, respectively.

segment length $\Delta\Gamma$ is computed from the apertures as follows.

$$\Delta\Gamma_{i,j,k} = \left[(A_{i+\frac{1}{2},j,k} - A_{i-\frac{1}{2},j,k})^2 \Delta y \Delta z + (A_{i,j+\frac{1}{2},k} - A_{i,j-\frac{1}{2},k})^2 \Delta x \Delta z + (A_{i,j,k+\frac{1}{2}} - A_{i,j,k-\frac{1}{2}})^2 \Delta x \Delta y \right]^{\frac{1}{2}} \tag{23}$$

Geometrical reconstruction with seven pyramids yields the volume fraction α.

$$\begin{aligned} \alpha_{i,j,k} = \frac{1}{3} \frac{1}{\Delta x \Delta y \Delta z} \Bigg[& A_{i+\frac{1}{2},j,k} \Delta y \Delta z \frac{1}{2} \Delta x + A_{i-\frac{1}{2},j,k} \Delta y \Delta z \frac{1}{2} \Delta x + A_{i,j+\frac{1}{2},k} \Delta x \Delta z \frac{1}{2} \Delta y \\ & + A_{i,j-\frac{1}{2},k} \Delta x \Delta z \frac{1}{2} \Delta y + A_{i,j,k-\frac{1}{2}} \Delta x \Delta y \frac{1}{2} \Delta z + A_{i,j,k+\frac{1}{2}} \Delta x \Delta y \frac{1}{2} \Delta z + \Delta\Gamma_{i,j,k} \phi_{i,j,k} \Bigg] \end{aligned} \tag{24}$$

Note that the described approach yields the volume fraction and apertures with respect to the positive phase. The values of the negative phase can be obtained from relations (16).

The level-set field is advected by the interface velocity $\mathbf{u}_I = \mathbf{n}_I u_I$ by solving the level-set advection equation.

$$\frac{\partial \phi}{\partial t} + \mathbf{u}_I \cdot \nabla \phi = 0 \tag{25}$$

The spatial term in Equation (25) is discretized using high-order upstream central (HOUC) [59] stencils. For the temporal integration, we apply the same scheme that is used to integrate the conservative variables, which typically is a Runge-Kutta method, see Subsection 3.2. The level-set field is only advected within a narrowband around the interface.

In order to apply the reconstruction stencils used in the finite-volume discretization near the interface, we extrapolate the real fluid state to the other side of the interface. We denote the cells on the other side of the interface as ghost cells. An arbitrary quantity ψ is extrapolated from the real cells to the ghost cells by advancing it in a fictitious time τ to steady state according to

$$\frac{\partial \psi}{\partial \tau} \pm \mathbf{n}_I \cdot \nabla \psi = 0. \tag{26}$$

The sign of the interface normal $\pm\mathbf{n}_I$ depends on the sign of the present phase: To extrapolate the real fluid state of the positive phase into its ghost cells, we must extrapolate into the direction of the negative level-set, i.e., $-\mathbf{n}_I$, and vice versa. The spatial term is discretized using a first-order upwind stencil. Temporal integration is performed with the Euler method. Note that we also use this extension procedure to extend the interface velocity from the cut cells into the narrowband around the interface, where we advect the level-set field.

The computation of the geometrical quantities requires the level-set field to be a signed distance function. During a simulation, the level-set field loses its signed distance property due to numerical errors and/or a shearing flow field. Additionally, since we only advect the level-set field within a narrowband around the interface, the level-set field develops a kink at the edge of the narrowband and the remainder of the computational domain. The signed distance property is maintained via reinitialization of the level-set field. The reinitialization equation reads

$$\frac{\partial \phi}{\partial \tau} + sgn(\phi^0)(|\nabla \phi| - 1) = 0. \tag{27}$$

Here, ϕ^0 represents the level-set at a fictitious time $\tau = 0$. We apply first-order [60] or higher order WENO-HJ [61] schemes to solve this equation. We reinitialize the level-set each physical time step for a fixed amount of fictitious time steps resulting in a sufficiently small residual of Equation (27).

The presented level-set method is not consistent when the interface crosses a cell face within a single time step, i.e., when new fluid cells are created or fluid cells have vanished. Figure 3 displays this scenario for a 1D discretization. The interface is moving from cell $i-1$ to cell i. At t^n, cell i is a newly created cell with respect to phase 1, and cell $i-1$ is a vanished cell with respect to phase

2. To maintain conservation, we must do the following. For phase 1, conservative variables must be taken from cell $i-1$ and put into the newly created cell i. For phase 2, conservative variables must be taken from cell i and put into the vanished cell $i-1$. In addition to the scenario where an interface crosses the cell face, small cut cells may generally lead to an unstable integration using the time step restriction that is based on a full cell. We apply a mixing procedure [56] that deals with both of these problems. The procedure is applied to two types of cells.

1. Cells where $\alpha = 0$ after integration but $\alpha \neq 0$ before (vanished cells).
2. Cells with $\alpha < \alpha_{\text{mix}}$ after integration (newly created cells and small cells).

We use a mixing threshold of $\alpha_{\text{mix}} = 0.6$. For each cell that requires mixing, we identify target (trg) cells from the interface normal. Consider cell i in Figure 3, which is a small/newly created cell for phase 1. Here, the target cell in x-direction is cell $i-1$, as $\mathbf{n}_{I1} \cdot \mathbf{i} < 0$. Analogously, cell $i-1$ is a vanished cell for phase 2. The corresponding target is cell i, since $\mathbf{n}_{I2} \cdot \mathbf{i} > 0$. In 3D, there are 7 target cells in total: One in each spatial direction x, y, and z, one in each plane xy, xz, and yz, and one in xyz. Seven mixing weights are computed as

$$\begin{aligned}
\beta_x &= |\mathbf{n}_I \cdot \mathbf{i}|^2 \alpha_{trg}, \\
\beta_y &= |\mathbf{n}_I \cdot \mathbf{j}|^2 \alpha_{trg}, \\
\beta_z &= |\mathbf{n}_I \cdot \mathbf{k}|^2 \alpha_{trg}, \\
\beta_{xy} &= |\left(\mathbf{n}_I \cdot \mathbf{i}\right)\left(\mathbf{n}_I \cdot \mathbf{j}\right)| \alpha_{trg}, \\
\beta_{xz} &= |\left(\mathbf{n}_I \cdot \mathbf{i}\right)\left(\mathbf{n}_I \cdot \mathbf{k}\right)| \alpha_{trg}, \\
\beta_{yz} &= |\left(\mathbf{n}_I \cdot \mathbf{j}\right)\left(\mathbf{n}_I \cdot \mathbf{k}\right)| \alpha_{trg}, \\
\beta_{xyz} &= |\left(\mathbf{n}_I \cdot \mathbf{i}\right)\left(\mathbf{n}_I \cdot \mathbf{j}\right)\left(\mathbf{n}_I \cdot \mathbf{k}\right)|^{2/3} \alpha_{trg}.
\end{aligned} \tag{28}$$

Here, α_{trg} denotes the volume fraction of the target cell in the corresponding direction. We normalize the mixing weights so that $\sum_{trg} \beta_{trg} = 1$, where $trg \in \{x, y, z, xy, xz, yz, xyz\}$. Subsequently, the mixing flux $\mathbf{M}_{trg}$ is computed like

$$\mathbf{M}_{trg} = \frac{\beta_{trg}}{\alpha\beta_{trg} + \alpha_{trg}} \left[(\alpha_{trg}\bar{\mathbf{U}}_{trg})\alpha - (\alpha\bar{\mathbf{U}})\alpha_{trg}\right]. \tag{29}$$

The conservative variables are then updated according to

$$\alpha\bar{\mathbf{U}} = \left(\alpha\bar{\mathbf{U}}\right)^* + \sum_{trg} \mathbf{M}_{trg}, \tag{30}$$

$$\alpha_{trg}\bar{\mathbf{U}}_{trg} = \left(\alpha_{trg}\bar{\mathbf{U}}_{trg}\right)^* - \sum_{trg} \mathbf{M}_{trg}. \tag{31}$$

Here, $\alpha\bar{\mathbf{U}}$ and $\alpha_{trg}\bar{\mathbf{U}}_{trg}$ denote the conservative variables of the cells that require mixing and the conservative variables of the corresponding target cells, respectively. Star-quantities denote conservative variables before mixing.

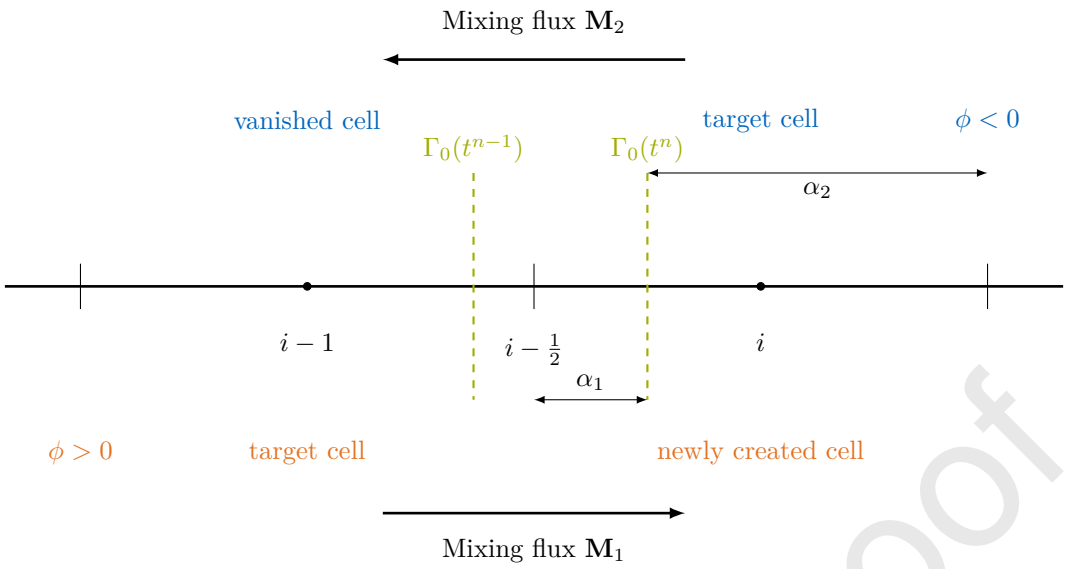


Figure 3: Schematic illustrating the mixing procedure in a 1D discretization at t^n. Orange and blue colors indicate the positive and negative phases, respectively. Green indicates interface positions Γ_0 at t^{n-1} and t^n.

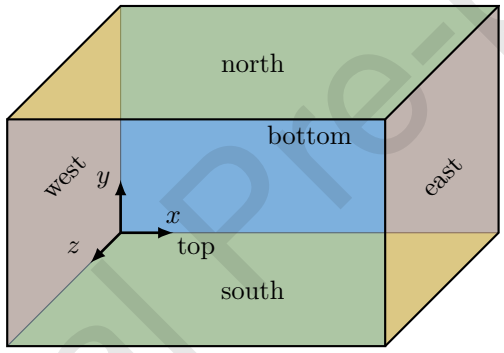


Figure 4: Computational domain with boundary locations.

3.7. Computational Domain and Boundary Conditions

The computational domain is a cuboid. Figure 4 depicts an exemplary computational domain including the nomenclature for the boundary locations. The solver provides symmetry, periodic, no-slip wall, Dirichlet, and Neumann boundary conditions. The no-slip wall boundary condition allows the user to specify either a constant value or time-dependent function for the wall velocity. The Dirichlet and Neumann boundary conditions allow the user to specify either a constant value or a space- and time-dependent function. Furthermore, (in 2D only) multiple different types of boundary conditions may be imposed along a single boundary location. Here, the user must specify the starting and end point of each of the different boundary types along the specific boundary location. The level-set implementation allows for arbitrary immersed solid boundaries.

Time Integration	Euler	
	TVD-RK2 [50]	
	TVD-RK3 [50]	
Flux Function/Riemann Solver	Lax-Friedrichs (LxF)	According to [17]
	Local Lax-Friedrichs (LLxF, Rusanov)	According to [17]
	HLL/HLLC/HLLC-LM [53, 17, 54, 62, 63]	Signal speed estimates see below
	AUSM+ [18]	
	Componentwise LLxF	Flux-splitting formulation
	Roe [15]	Flux-splitting formulation
Signal Speed Estimates	Arithmetic	
	Davis [64]	
	Einfeldt [65]	
	Toro [17]	
Spatial Reconstruction	WENO1 [49]	
	WENO3-JS/Z/N/F3+/NN [49, 66, 67, 37]	
	WENO5-JS/Z [49, 68]	
	WENO6-CU/CUM [69, 70]	
	WENO7-JS [71]	
	WENO9-JS [71]	
	TENO5 [72]	
	Second-order central	For dissipative terms only
	Fourth-order central	For dissipative terms only
Spatial Derivatives	Second-order central	
	Fourth-order central	
	HOUC-3/5/7 [59]	
Levelset reinitialization	First-order [60]	
	HJ-WENO [61]	
Ghost fluid extension	First-order upwind [56]	
LES Modules	ALDM [73]	
Equation of State	Ideal Gas	
	Stiffened Gas [74]	
	Tait [75]	
Boundary Conditions	Periodic	
	Zero Gradient	E.g., used for outflow boundaries
	Neumann	E.g., for a prescribed heat flux
	Dirichlet	
	No-slip Wall	
	Immersed Solid Boundaries	Arbitrary geometries via level-set

Table 2: Overview on numerical methods available in JAX-Fluids.

4. Software Implementation Details

In the past, CFD solvers have been written predominantly in low-level programming languages like Fortran and C/C++. These languages offer computational performance and CPU parallelization

capabilities. However, the integration of ML models, which are typically coded in Python, is not straightforward and automatic differentiation capabilities are virtually nonexistent. With the present work, we want to provide a state-of-the-art CFD framework for complex flow physics that allows for seamless integration and end-to-end optimization of ML models, without major performance losses.

There are three major differentiable programming packages in Python which fulfill these requirements: TensorFlow [25], JAX [27], and PyTorch [26]. While TensorFlow leans more towards large-scale ML research, PyTorch and JAX have become especially popular for ML research in the physical sciences. In particular, recent works have shown that JAX can be used successfully in the context of CFD, e.g., [46, 2]. As JAX additionally offers an explicit NumPy API for scientific computing, we use JAX as the fundamental building block of JAX-Fluids.

In this section, we give implementation details and an overview of the algorithmic structure of JAX-Fluids.

4.1. Array Programming in JAX

JAX is a Python library for high-performance numerical computations, which uses XLA to compile and run code on accelerators like CPUs, GPUs, and TPUs. Designed as a machine learning library, JAX supports automatic differentiation [12]. In particular, JAX comes with a fully-differentiable version of the popular NumPy package [76] called JAX NumPy. NumPy provides high-level functionality to handle large arrays and matrices. Both Python and NumPy are widely popular and easy to use.

The backbone of (JAX) NumPy is the multidimensional array, so called `jax.numpy.DeviceArray`. We use arrays to store all of our field data. In particular, the vectors of conservative and primitive variables, $\mathbf{U}$ and $\mathbf{W}$, are stored in arrays of shape $(5, N_x + 2N_h, N_y + 2N_h, N_z + 2N_h)$. (N_x, N_y, N_z) denotes the resolution in the three spatial directions, and N_h specifies the number of halo cells. Our implementation naturally degenerates for one- and two-dimensional settings by using only a single cell in the excess dimensions.

In the *array programming paradigm* (see [77] for details on numerical operations on arrays in NumPy), every operation is performed on the entire array. I.e., instead of writing loops over the array as is common in Fortran or C/C++, we use appropriate indexing/slicing operations. As a result, many parts of the code used from classical CFD solvers have to be rewritten according to the *array programming paradigm*. As an example we include the source code of our implementation of the second-order central cell face reconstruction scheme in Figure 5. The `class CentralSecondOrderReconstruction` inherits from `class SpatialReconstruction`. This parent class has an abstract member function `reconstruct_xi`, which must be implemented in every child class. `reconstruct_xi` receives the entire `buffer` and the reconstruction direction `axis` as arguments.

The data buffer has to be indexed/sliced differently depending on the spatial direction of the reconstruction and the dimensionality of the problem. Note that the buffer array has shape $(5, N_x + 2N_h, N_y + 2N_h, N_z + 2N_h)$ if the problem is three-dimensional, $(5, N_x + 2N_h, N_y + 2N_h, 1)$ if the problem

```python
from functools import partial
import jax, jax.numpy as jnp
from jaxfluids.stencils.spatial_reconstruction import SpatialReconstruction

class CentralSecondOrderReconstruction(SpatialReconstruction):

    def __init__(self, nh: int, inactive_axis: jnp.array) -> None:
        super(CentralSecondOrderReconstruction, self).__init__(nh=nh, inactive_axis=inactive_axis)
        self.slices = [
            [   jnp.s_[..., self.nh-1:-self.nh  , self.nhy, self.nhz],         #  X-DIRECTION
                jnp.s_[..., self.nh  :-self.nh+1, self.nhy, self.nhz],   ],

            [   jnp.s_[..., self.nhx, self.nh-1:-self.nh  , self.nhz],         #  Y-DIRECTION
                jnp.s_[..., self.nhx, self.nh  :-self.nh+1, self.nhz],   ],

            [   jnp.s_[..., self.nhx, self.nhy, self.nh-1:-self.nh  ],         #  Z-DIRECTION
                jnp.s_[..., self.nhx, self.nhy, self.nh  :-self.nh+1],   ],
        ]

    @partial(jax.jit, static_argnums=(0, 2))
    def reconstruct_xi(self, buffer: jnp.array, axis: int) -> jnp.array:
        s_ = self.slices[axis]
        cell_face_state_xi = 0.5 * ( buffer[s_[0]] + buffer[s_[1]] )
        return cell_face_state_xi
```

Figure 5: Code snippet for the second-order central cell face reconstruction stencil.

is two-dimensional, and $(5, N_x + 2N_h, 1, 1)$ if the problem is one-dimensional. The slice indices have to be adapted accordingly. To prevent boilerplate code in the reconstruction routine, we abstract the slicing operations. The member variable `self.slices` is a list that holds the correct slice objects for each spatial direction. Consider reconstructing in x-direction (`axis=0`) using the present second-order cell face reconstruction scheme,

$$\mathbf{U}_{i+\frac{1}{2},j,k} = \frac{1}{2}(\mathbf{U}_{i,j,k} + \mathbf{U}_{i+1,j,k}). \tag{32}$$

Here, we require two slice objects: `jnp.s_[...,self.nh-1:-self.nh,self.nhy,self.nhz]` for $\mathbf{U}_{i,j,k}$ and `jnp.s_[...,self.nh:-self.nh+1,self.nhy,self.nhz]` for $\mathbf{U}_{i+1,j,k}$. The variable `self.nh` denotes the number of halo cells. The slices in x-direction, i.e., `self.nh-1:-self.nh` and `self.nh:-self.nh+1`, are determined by the reconstruction scheme itself. `self.nhy` and `self.nhz` denote the slice objects for the dimensions in which we do not reconstruct. These are either `self.nh:-self.nh` if the dimension is active or `None:None` if the dimension is inactive. `self.nhx`, `self.nhy`, and `self.nhz` are defined in the parent class.

4.2. Object-oriented Programming in the Functional Programming World of JAX

As already alluded to in the previous subsection, we use object-oriented programming (OOP) throughout the entire JAX-Fluids solver. Although JAX leans inherently more towards a functional

programming style, we have opted to program JAX-Fluids in a modular object-oriented approach since this has several benefits. Firstly, comprehensive CFD solvers typically offer a plethora of interchangeable numerical algorithms. Naturally, OOP allows the implementation of many derived classes, e.g., different spatial reconstructors (see `class SpatialReconstruction` in the previous subsection), and saves boilerplate code. Secondly, the modular OOP approach allows users to customize a solver specific to their problem. Via a numerical setup file, the user can detail the numerical setup prior to every simulation. Thirdly, we want to stress that the modularity of our solver allows for straightforward integration of custom modules and implementations. For example, avid ML-CFD researchers can easily implement their own submodule into the JAX-Fluids framework, either simply for forward simulations or for learning new routines from data.

4.3. Just-in-time (jit) Compilation and Pure Functions

JAX offers the possibility to just-in-time (jit) compile functions, which significantly increases the performance. However, jit-compilation imposes two constraints:

1. **The function must be a pure function**. A function is pure if its return values are identical for identical input arguments and the function has no side effects.
2. **Control flow statements in the function must not depend on input argument values**. During jit-compilation, an abstract version of the function is cached that works for arbitrary argument values. I.e., the function is not compiled for concrete argument values but rather for the set of all possible argument values where only array shape and type are fixed. Therefore, control flow statements that depend on the input argument values can not be jit-compiled, unless the argument is a *static argument*. In this case, the function is recompiled for all values that the *static argument* takes during runtime.

The aforementioned requirements for jit-compilation have the following impact on our implementation:

- The `self` argument in jit-compiled member functions must be a *static argument*. This implies that class member variables that are used in the function are static and hence must not be modified. In other words, the type of class member variables is similar to the C++ type `constexpr`.
- Control flow statements in jit-compiled functions can only be evaluated on *static arguments*. In our code, there are three distinct types of control flow statements where this has an impact:
    1. **Conditional exit of a for/while loop**. The top level compute loop over the physical simulation time (compare Algorithm 1) is not jit-compiled, since it consists of a while loop that is exited when the final simulation time is reached $t \geq t_{end}$. However, t is not a static variable. We therefore only jit-compile the functions `compute_timestep`, `do_integration_step`, and `compute_forcings`. These are the functions that constitute the heavy compute within the main loop.

```python
from functools import partial
import jax, jax.numpy as jnp

class LevelsetHandler():
    def __init__(self, ...) -> None:
        pass

    @partial(jax.jit, static_argnums=(0, 3))
    def compute_levelset_advection_rhs_xi(self, levelset: jnp.array, interface_velocity: jnp.array,
            axis: int) -> jnp.array:

        # LEFT AND RIGHT SIDED DERIVATIVE
        derivative_L = self.spatial_derivative.derivative_xi(levelset, self.cell_sizes[axis], axis, 0)
        derivative_R = self.spatial_derivative.derivative_xi(levelset, self.cell_sizes[axis], axis, 1)

        # UPWINDING DEPENDING ON LOCAL INTERFACE VELOCITY
        velocity    = interface_velocity[axis]
        mask_L      = jnp.where(velocity >= 0.0, 1.0, 0.0)
        mask_R      = 1.0 - mask_L

        # RIGHT-HAND-SIDE EVALUATION
        rhs_contribution = - velocity * (mask_L * derivative_L + mask_R * derivative_R)

        return rhs_contribution
```

Figure 6: Code snippet showing the right-hand-side computation of the level-set advection equation.

2. **Conditional slicing of an array**. In multiple parts of the JAX-Fluids code, arrays are sliced depending on the present spatial direction. The present spatial direction is indicated by the input argument `axis` (compare cell face reconstruction in Figure 5). The code that is actually executed conditionally depends on the value of `axis`. Therefore, `axis` must be a static argument. Functions that receive `axis` as an input argument are compiled for all values that `axis` might take during runtime. Each compiled version of those functions is cached.
3. **Conditional executation of code sections**. We explained above that class member variables are always static. In practice, we often use them to conditionally compile code sections, much like the `if constexpr` in C++. An example is the cell face reconstruction in the convective flux calculation. The reconstruction can be done on primitive or conservative variables in either physical or characteristic space.

- Element-wise conditional array operations are implemented using masks, as jit-compilation requires the array shapes to be fixed at compile time. Figure 6 exemplarily illustrates the use of masks, here for the evaluation of the level-set advection equation. We make frequent use of `jnp.where` to implement element-wise conditional array operations.

```python
import json
from jaxfluids import InputReader, Initializer, SimulationManager

numerical_setup_dict    = json.load(open("numerical_setup.json"))
case_setup_dict         = json.load(open("case_setup.json"))

input_reader        = InputReader(case_setup_dict, numerical_setup_dict)
initializer         = Initializer(input_reader)
simulation_manager  = SimulationManager(input_reader)

initial_buffer      = initializer.initialization()
simulation_manager.simulate(initial_buffer)
```

Figure 7: Code snippet illustrating how to run a simulation with JAX-Fluids.

4.4. Main Objects and Compute Loops

We put emphasis on making JAX-Fluids an easy-to-use Python package for ML-CFD research. Figure 7 shows the required lines of code to run a simulation. The user must provide a numerical setup and a case setup file in the *json* format. The numerical setup specifies the combination of numerical methods that will be used for the simulation. The case setup details the physical properties of the simulation including the spatial domain and its resolution, initial and boundary conditions, and material properties. As an example, we include the numerical setup and case setup for the Sod shock tube test case (see Section 5.1.2) in the appendix in Figure A.22 and A.23, respectively. Using these files, we create a `class InputReader` instance, which we denote as `input_reader`. The `input_reader` performs necessary data type transformations and a thorough sanity check ensuring that the provided numerical and case setups are consistent. Then, a `class Initializer` and a `class SimulationManager` object are created using the `input_reader` object. The `class Initializer` implements functionality to generate the initial buffers from either the initial condition specified in the case setup file or from a restart file. We receive these buffers by calling the `initialization` method of the `initializer` object. The `class SimulationManager` is the main class in JAX-Fluids, implementing the algorithm to advance the initial buffers in time using the specified numerical setup. The initial buffers must be passed to the `simulate` method of the `simulation_manager` object. The simulation starts upon execution of this function.

The code contains three major loops:

1. **Loop over physical simulation time**, see Algorithm 1.
2. **Loop over Runge-Kutta stages**, see Algorithm 2.
3. **Loop over active spatial dimensions**, see Algorithm 3.

```
while time < end time do
    compute_timestep()
    forcings_handler.compute_forcings()
    do_integration_step()
    output_writer.write_output()
end
```

Algorithm 1: Loop over physical simulation time in the `simulate` function. Orange text color indicates functions that are only executed for simulations with active forcings.

```
for RK stages do
    space_solver.compute_rhs()
    levelset_handler.transform_volume_averages_to_conservatives()
    time_integrator.prepare_buffers_for_integration()
    time_integrator.integrate_conservatives()
    time_integrator.integrate_levelset()
    levelset_handler.reinitialize_levelset()
    boundary_condition.fill_boundaries_levelset()
    levelset_handler.compute_geometrical_quantities()
    levelset_handler.mix_conservatives()
    levelset_handler.transform_conservatives_to_volume_averages()
    get_primitives_from_conservatives()
    levelset_handler.extend_primitives_into_ghost_cells()
    boundary_condition.fill_material_boundaries()
end
```

Algorithm 2: Loop over Runge-Kutta stages in the `do_integration_step` function. Blue text color indicates functions that are only executed for two-phase simulations.

```
for active axis do
    flux_computer.compute_inviscid_flux_xi()
    flux_computer.compute_viscous_flux_xi()
    flux_computer.compute_heat_flux_xi()
    levelset_handler.weight_cell_face_flux_xi()
    levelset_handler.compute_interface_flux_xi()
    levelset_handler.compute_levelset_advection_rhs()
end
```

Algorithm 3: Loop over spatial dimensions in the `compute_rhs` function. Blue text color indicates functions that are only executed for two-phase simulations.

4.5. Gradient Computation in JAX-Fluids

JAX-Fluids offers the `simulate` method to perform a standard forward CFD simulation. In this regard, JAX-Fluids serves as a physics simulator for data generation, development of numerical methods, and exploration of fluid dynamics. The `simulate` method does not have a return value, therefore, it is not meant to be used for end-to-end optimization.

To use the automatic differentiation capabilities of JAX-Fluids, we offer the `feed_forward` method. This method takes in a batch of initial buffers and propagates them in time for a fixed number of time steps. The user provides the number of integration steps and a fixed time step. The output of the `feed_forward` method is the solution trajectory. In particular, the shape of the initial buffer is $(N_b, 5, N_x, N_y, N_z)$ where N_b is the batch size. The shape of the solution trajectory is $(N_b, N_T + 1, 5, N_x, N_y, N_z)$ where N_T is the number of integration steps. Internally, the `feed_forward` method uses the `jax.vmap` routine to vectorize over the batch dimension. `feed_forward` is jit-compilable and can be differentiated with `jax.grad` or `jax.value_and_grad`. The `feed_forward` method of JAX-Fluids can therefore be used for end-to-end optimization of ML models.

4.6. Integration of ML models into JAX-Fluids

JAX-Fluids works with Haiku [78] and Optax [79]. The Haiku package is a neural network library for JAX. In Haiku, neural networks are of type `haiku.Module`. To use them in combination with JAX, the feedforward method of the network has to be embedded in a function that is transformed into a forward wrapper object of type `haiku.Transformed`. This forward wrapper object provides two pure methods, `init` and `apply`. The `init` method initializes the network parameters, and the `apply` method executes the feedforward of the network. Network parameters have to be explicitly passed to the `apply` method. We refer to the Haiku documentation for more details. Optax provides optimization routines, e.g., the popular Adam optimizer [80].

In JAX-Fluids, we provide functionality to include preexisting ML models and optimize new ones. Neural networks can be passed to the `simulate` and `feed_forward` method. Note that only the `feed_forward` method can be differentiated, see Subsection 4.5. I.e., it must be used for optimization of deep learning models. A typical use case in ML-CFD research is substituting a conventional numerical subroutine with a data-driven alternative. We provide a number of interfaces inside the JAX-Fluids solver to which network modules can be passed from `feed_forward`, e.g., to the cell face reconstruction, the Riemann solver, or the forcing module. On the top level, the user gives a dictionary of transformed network modules and a dictionary with corresponding network parameters to the `feed_forward` method. The keys of these dictionaries specify the JAX-Fluids subroutine to which the corresponding values are passed.

5. Validation of JAX-Fluids as a Classical CFD Simulator

We show the capabilities of JAX-Fluids as a classical fluid dynamics simulator. We validate our implementation on established one- and two-dimensional test cases from gas dynamics and several canonical turbulent flows. In Subsection 5.1 we first validate the single-phase implementation in JAX-Fluids. In Subsection 5.2 we then validate the two-phase level-set implementation for fluid-solid and fluid-fluid interactions.

We define two numerical setups which we will use predominantly throughout this section. Firstly, we use the *High-Order Godunov* formulation. This setup consists of a WENO5-JS reconstruction of the primitive variables with an approximate HLLC Riemann solver. We will refer to this setup as *HLLC*. Secondly, we employ the *Flux-Splitting* formulation. In particular, we choose Roe approximate Riemann solver with a WENO5-JS flux reconstruction in characteristic space. This setup will be denoted as *ROE*. The dissipative fluxes are discretized with a fourth-order central finite difference stencil as described in Section 3.4. If not specified otherwise we will use the stiffened gas equation of state with $\gamma = 1.4$ and $B = 0$, i.e., the ideal gas law. We will use the classical TVD-RK3 scheme with $CFL_{\text{conservatives}} = 0.9$ for time integration. For the two-phase test cases, we additionally apply the following methods. The level-set advection equation is discretized with a HOUC5 stencil. We solve the extension equation using a first-order upwind spatial derivative stencil combined with an Euler integration scheme. Here, we apply a fixed number of 15 steps with $CFL_{\text{extension}} = 0.7$. The reinitialization equation is solved with a WENO3-HJ stencil combined with a TVD-RK2 scheme. The level-set field is reinitialized each physical time step by integrating the reinitialization equation for one step with $CFL_{\text{reinitialization}} = 0.7$. We refer to Section 3 and therein to Table 2 for details on numerical models.

5.1. Single Phase Simulations

5.1.1. Convergence Study

We analyze the convergence behavior of our solver. We simulate the advection of a density profile in a one-dimensional periodic domain of extent $x \in [0, 1]$ with constant velocity $u = 1$ and pressure $p = 1$. We use a sinusoidal initial condition for the density

$$\rho(x, t = 0) = 1.5 + \sin(2\pi x). \tag{33}$$

Note that we initialize the cell-averaged values of Equation (33). I.e., for cell i with cell-center x_i we use $\bar{\rho}_i = 1.5 - 1.0/(2\pi\Delta x)(\cos(2\pi x_{i+1/2}) - \cos(2\pi x_{i-1/2}))$. We conduct simulations with WENO1-JS, WENO3-JS, and WENO5-JS spatial discretizations and evaluate the rate of convergence upon mesh refinement. We use TVD-RK2 for WENO1- and WENO3-JS. For WENO5-JS we use TVD-RK3. We use a fixed time step $\Delta t = 1 \times 10^{-4}$ which is chosen small enough to exclude any influence of the time integration scheme. The simulation is propagated until $t_{\text{end}} = 1.0$. Figure 8 shows the convergence

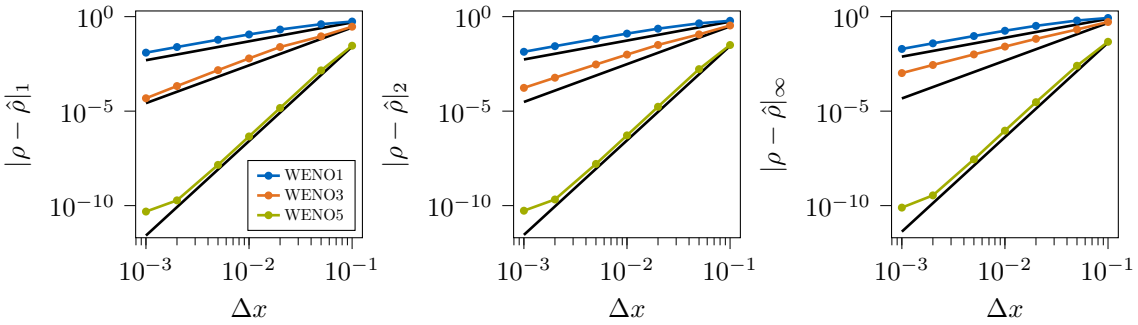


Figure 8: Error convergence $|\rho - \hat{\rho}|_p$ for the linear advection of Equation (33) with WENO1, WENO3, and WENO5 spatial discretization. $\hat{\rho}$ is the analytical solution at $t_{\text{end}} = 1.0$. From left to right: l_1, l_2, and l_∞ norms.

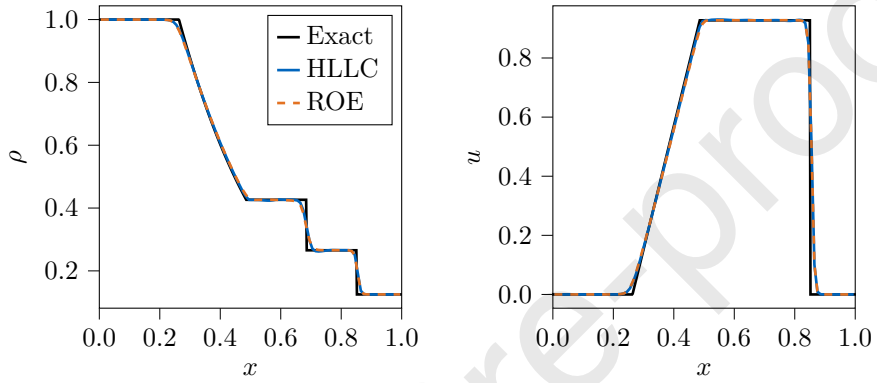


Figure 9: Sod shock tube: density ρ and velocity u at $t = 0.2$.

rates in the l_1, l_2, and l_∞ norms. We increase the resolution from 10 to 1000 points. The expected convergence rates of $\mathcal{O}(\Delta x^1)$, $\mathcal{O}(\Delta x^2)$, and $\mathcal{O}(\Delta x^5)$ are reached. Note that it is well known that the convergence order of WENO3-JS drops to second order under the presence of extreme points.

5.1.2. Shock Tube Tests

The shock tube tests of Sod [81] and Lax [82] are standard one-dimensional test cases for validating fluid solvers for compressible flows. Specifically, we use these test cases to validate the implementation of the convective fluxes and the shock-capturing schemes. In both shock tube test cases three waves emanate from the initial discontinuities: a left running rarefaction, a right running contact discontinuity, and a right running shock. A detailed description of the tests and their setups are provided in the presented references. We discretize the domain $x \in [0, 1]$ with $N = 100$ points. The analytical reference solution is taken from an exact Riemann solver (e.g, [54]). We run both shock tube tests with the *HLLC* and *ROE* setups. Figures 9 and 10 show the density and velocity distributions for the Sod and Lax shock tube tests at $t = 0.2$ and $t = 0.14$, respectively. The *HLLC* and *ROE* solutions agree very well with the analytical reference. The *HLLC* solutions show slightly oscillatory behavior which is due to the cell face reconstruction of the primitive variables, see [52].

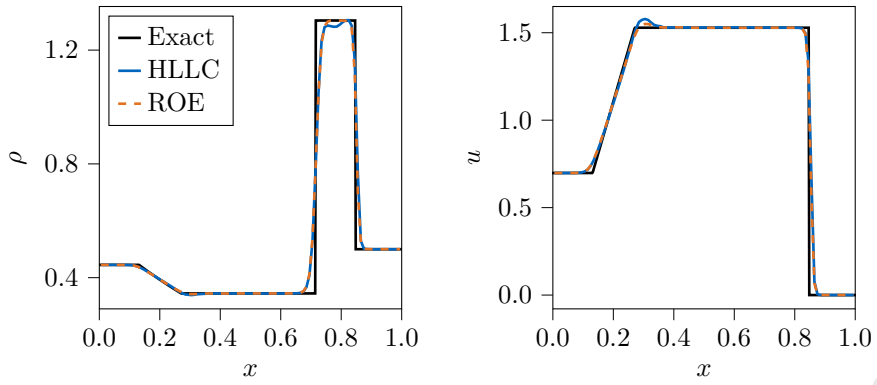


Figure 10: Lax shock tube: density ρ and velocity u at $t = 0.14$.

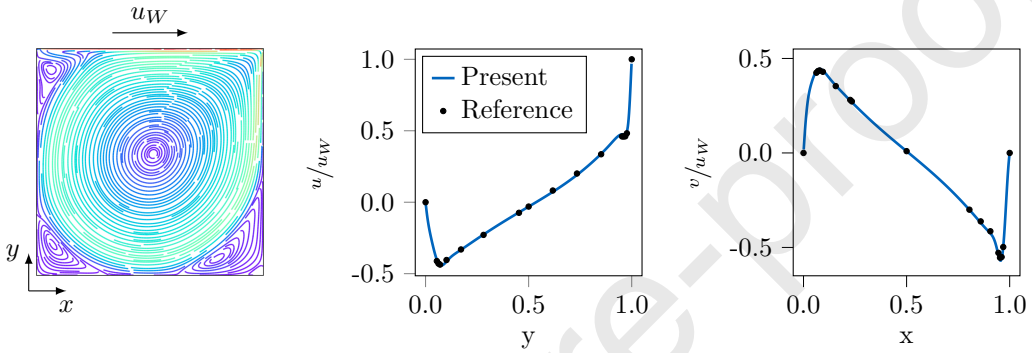


Figure 11: Lid driven cavity at $Re = {}^{u_W L}/_{\nu} = 5000$. (Left) Instantaneous streamlines with colors ranging from largest (red) to smallest (blue) value of the absolute velocity. (Middle) Normalized u velocity over y through domain center. (Right) Normalized v velocity over x through domain center. Reference is taken from [83].

5.1.3. Lid-driven Cavity

The lid-driven cavity test case describes the flow in a square cavity that is driven by a moving wall. Viscous forces lead to the generation of one primary vortex and, depending on the Reynolds number, one or more secondary vortices. We use this test case to validate the implementation of the viscous fluxes. The computational domain $x \times y \in [0, 1] \times [0, 1]$ is discretized with a grid consisting of 500×500 cells. All boundary conditions are no-slip walls. The north wall is moving with a constant velocity u_W, resulting in a Reynolds number $Re = \frac{u_W L}{\nu} = 5000$. The test case is simulated until a steady state is reached with the *HLLC* setup. Figure 11 depicts the distribution of $\frac{u}{u_W}$ over y and $\frac{v}{u_W}$ over x across the domain center. The present result agrees very well with the reference [83].

5.1.4. Compressible Decaying Isotropic Turbulence

Many flows in nature and engineering applications are turbulent. The direct numerical simulation (DNS) of turbulent flows requires the resolution of the smallest scales present (the Kolmogorov scales) which is prohibitively expensive [84]. In JAX-Fluids we have implemented the implicit large eddy simulation (ILES) model *ALDM* [73] which enables us to simulate complex compressible turbulent

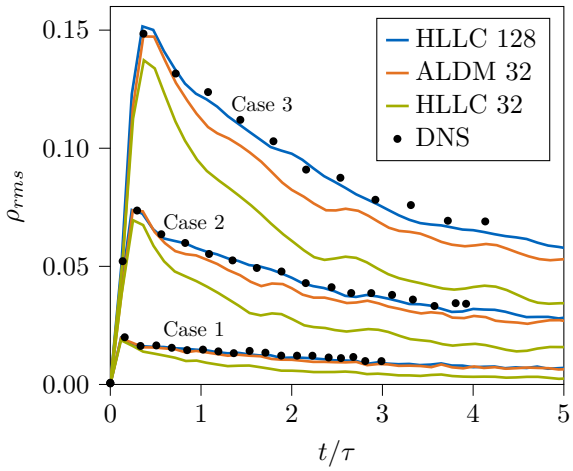


Figure 12: Temporal evolution of the rms density fluctuations for decaying isotropic turbulence. The time axis is normalized with the initial eddy turnover time. The DNS results are cases 1-3 from Spyropoulos et al. [85].

flows up to very high Reynolds numbers. To validate the implementation of *ALDM* and the viscous forces, we simulate compressible decaying isotropic turbulence at various turbulent Mach numbers. Specifically, we investigate the performance of the *ALDM* ILES implementation on the basis of cases 1-3 from Spyropoulos et al. [85]. The turbulent Mach numbers $M_t = \sqrt{< u^2 + v^2 + w^2 >}/ < c >$ for the three cases are 0.2, 0.4, and 0.6, respectively. The turbulent Reynolds number is $Re_T = \frac{\rho u'^4}{\epsilon \mu} = 2742$, where u' is the root-mean-square (rms) velocity and ϵ is the dissipation rate of turbulent kinetic energy [84]. The spatial domain has extent $x \times y \times z \in [0, 2\pi] \times [0, 2\pi] \times [0, 2\pi]$. We use the DNS data from [85] and an *HLLC* simulation with a resolution of 128^3 cells as reference. The LES simulations are performed on a coarse grid with 32^3 cells. We use *ALDM* and *HLLC* on this resolution to conduct LES simulations. The initial velocity field is divergence free and has the energy spectrum

$$E(k) = Ak^4 \exp(-2k^2/k_0^2), \tag{34}$$

where k is the wave number, k_0 is the wave number at which the spectrum is maximal, and A is a constant chosen to adjust a specified initial kinetic energy. The initial density and temperature fields are uniform. Figure 12 shows the temporal evolution of the rms density fluctuations $\rho_{rms} = \sqrt{< \rho' \rho' >}$. We normalize the time axis with the initial eddy turnover time $\tau = \lambda_f / u' \approx 0.85$. λ_f is the lateral Taylor microscale [84]. The *HLLC* simulation at 128^3 recovers the DNS reference very well. On the coarse mesh, the performance of the *ALDM* ILES becomes obvious when compared to *HLLC* simulations at the same resolution. *ALDM* gives good results consistent with the DNS data, indicating that *ALDM* recovers the correct sub-grid scale terms for compressible isotropic turbulence.

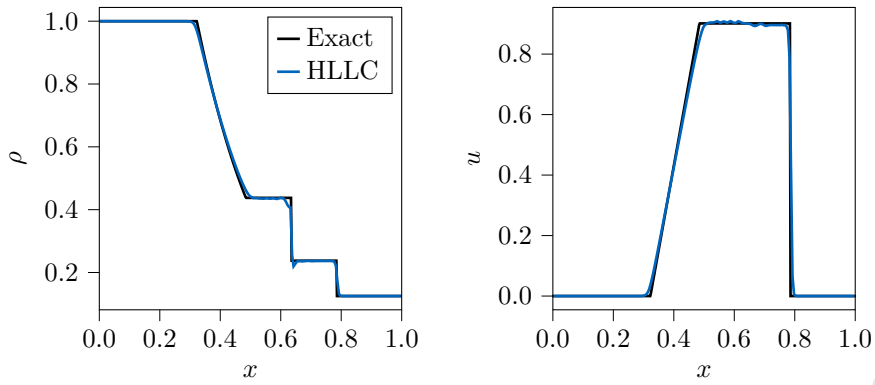


Figure 13: Air-helium shock tube problem. Density ρ and velocity u at $t = 0.15$.

5.2. Two-phase Simulations

5.2.1. Two-phase Sod Shock Tube

We consider a two-phase variant of the Sod shock tube problem [81]. This test case validates the inviscid fluid-fluid interface interactions. In particular, we investigate an air-helium shock tube problem in which the materials left and right of the initial discontinuity are air ($\gamma_{\text{air}} = 1.4$) and helium ($\gamma_{\text{helium}} = 1.667$), respectively. We use the previously described *HLLC* setup. The domain $x \in [0, 1]$ is resolved with 200 cells. Figure 13 shows the results at $t = 0.15$. The numerical approximations are in good agreement with the analytical solution. The interface position, shock speed, and shock strength are captured correctly. We observe slight density oscillations around the interface. This is in agreement with previous literature [56] as no isobaric fix is employed.

5.2.2. Bow Shock

Bow shocks occur in supersonic flows around blunt bodies [86]. Here, we simulate the flow around a stationary cylinder at high Mach numbers $Ma = \sqrt{\mathbf{u} \cdot \mathbf{u}} / \sqrt{\gamma \frac{p}{\rho}} = \{3, 20\}$. This test case validates the implementation of the inviscid fluid-solid interface fluxes. The computational domain $x \times y \in [-0.3, 0.0] \times [-0.4, 0.4]$ is discretized with a grid consisting of 480×1280 cells. A cylinder with diameter 0.2 is placed at the center of the east boundary. The north, east, and south boundaries are zero-gradient. The west boundary is of Dirichlet type, imposing the post shock fluid state. The fluid is initialized with the post shock state, i.e., $(\rho, u, v, p) = \left(1, \sqrt{1.4}Ma, 0, 1\right)$. We simulate the test case with the *HLLC* setup until a steady state solution is reached. Figure 14 illustrates the steady state density and pressure distributions. The results compare well to results from literature, e.g., [63].

5.2.3. Oscillating Drop

We consider a drop oscillating due to the interplay of surface tension and inertia. Starting from an ellipsoidal shape, surface tension forces drive the drop to a circular shape. This process is associated with a transfer of potential to kinetic energy. The oscillating drop test case validates the implementation of the surface tension forces. The drop oscillates with a distinct frequency. The oscillation period

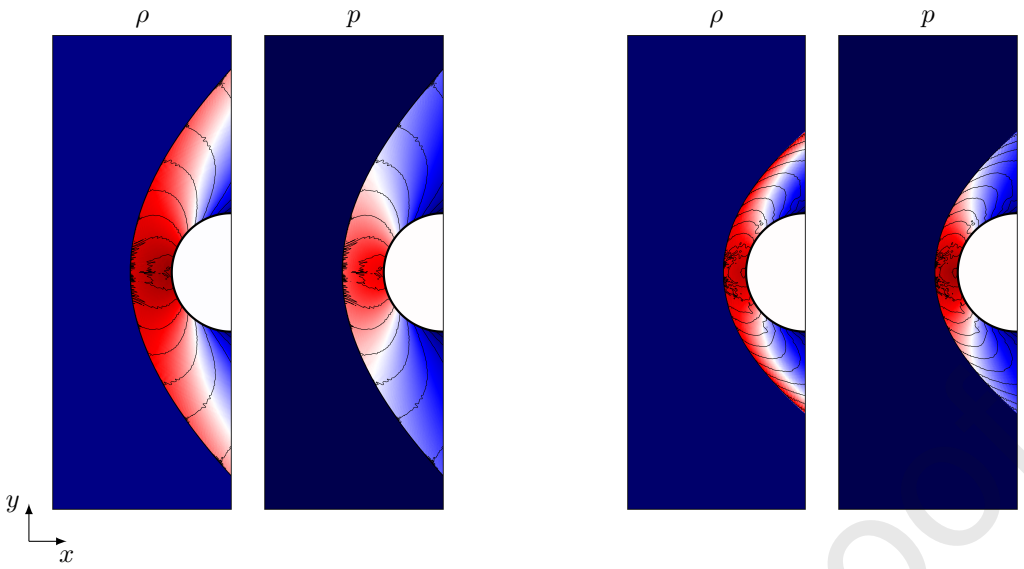


Figure 14: Density and pressure for the bow shock at $Ma = 3$ (left) and $Ma = 20$ (right). The colormap ranges from minimum (blue) to maximum (red) value; $\rho \in [0.7, 4.5]$, $p \in [1.0, 15.0]$ for $Ma = 3$ and $\rho \in [0.7, 7.2]$, $p \in [1.0, 550.0]$ for $Ma = 20$. The black lines represent Mach isocontours from 0.1 to 2.5 in steps of 0.2.

T is given by [87]

$$\omega^2 = \frac{6\sigma}{(\rho_b + \rho_d)R^3}, \qquad T = \frac{2\pi}{\omega}. \tag{35}$$

We discretize the computational domain $x \times y \in [0, 1] \times [0, 1]$ with a grid consisting of 200×200 cells. We place an ellipse with semi-major and semi-minor axes of 0.2 and 0.15 at the center of the domain. The effective circle radius is therefore $R = 0.17321$. The bulk and drop densities are $\rho_b = \rho_d = 1.0$ and the surface tension coefficient is $\sigma = 0.05$. All boundaries are zero-gradient. We use the *HLLC* setup for this simulation. Figure 15 displays instantaneous pressure fields and the kinetic energy of the drop over time. The present result for the oscillation period is $T = 1.16336$, which is in very good agreement with the analytical reference $T_{ref} = 1.16943$.

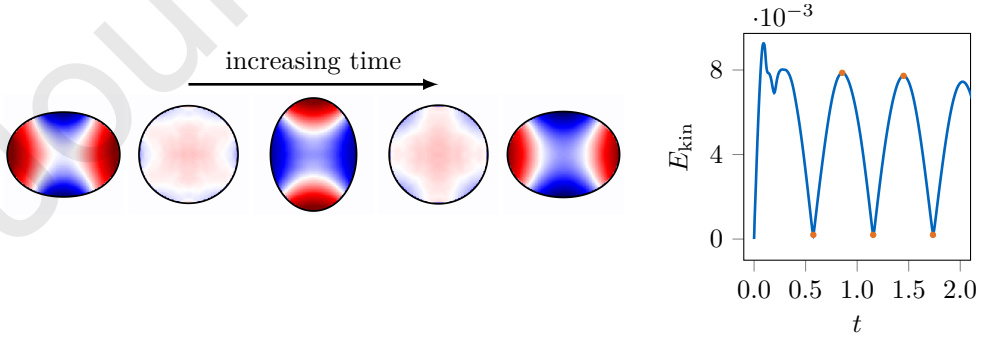


Figure 15: Oscillating drop. (Left) Temporal evolution of the pressure distribution within the drop. The colors range from maximum (red) to minimum (blue) value within the shown time period. (Right) Kinetic energy $E_{\text{kin}} = \int_V \rho \mathbf{u} \cdot \mathbf{u} dV$ of the drop over time. The orange dots indicate the times that correspond to the pressure distributions on the left.

5.2.4. Shear Drop Deformation

The shear drop deformation test case describes the deformation of an initially circular shaped drop due to homogenous shear. Viscous forces lead to the deformation to an ellipsoidal shape. For stable parameters, surface tension forces will eventually balance the viscous forces, which results in a steady state solution. The shear flow is generated with moving no-slip wall boundaries. We use this test case to validate the viscous fluid-fluid interface fluxes. The steady state solution is characterized by the viscosity ratio μ_b/μ_d and the capillary number Ca

$$Ca = \frac{\mu_b R \dot{s}}{\sigma}, \tag{36}$$

where R denotes the initial drop radius, σ the surface tension coefficient, $\dot{s}$ the shear rate, and μ_b and μ_d the viscosities of the drop and bulk fluid, respectively. The following relation holds for small deformations [88].

$$D = \frac{B_1 - B_2}{B_1 + B_2} = Ca \frac{19\mu_b/\mu_d + 16}{16\mu_b/\mu_d + 16} \tag{37}$$

Herein, B_1 and B_2 indicate the semi-major and semi-minor axes of the steady state ellipse. To simulate this test case, we discretize the domain $x \times y \in [0,1] \times [0,1]$ with a grid that consists of 250×250 cells. A drop with radius $R = 0.2$ is placed at the center of the domain. We move the north and south wall boundaries with an absolute velocity of $u_W = 0.1$ in positive and negative direction, respectively. This results in a shear rate of 0.2. At the east and west boundaries we enforce periodicity. The viscosities are $\mu_b = \mu_d = 0.1$. We simulate multiple capillary numbers by varying the surface tension coefficient with the *HLLC* setup until a steady state solution is reached.

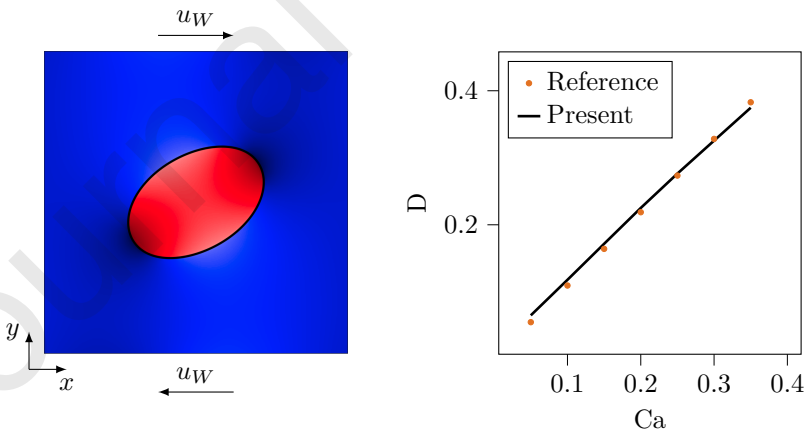
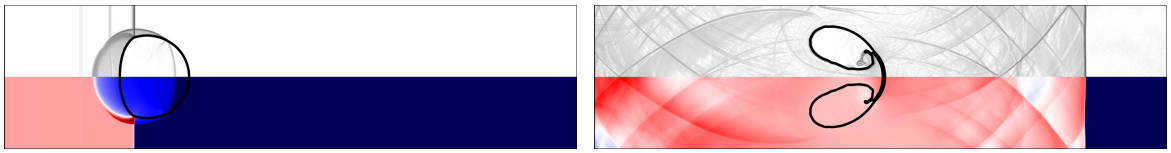


Figure 16: Shear drop deformation. (Left) Steady state pressure field for $Ca = 0.2$. (Right) Deformation parameter D over Ca.

Figure 16 illustrates the pressure distribution of the steady state ellipse at $Ca = 0.2$. Furthermore, it shows the deformation parameter D over the capillary number Ca. The present result agrees well with the analytical reference.



$t = 55\mu s$ $t = 555\mu s$

Figure 17: Visualizations of the flow field at two different instances in time. The upper half of each image shows numerical schlieren, the lower half shows the pressure field from the smallest (blue) to the largest (red) pressure value. The black line indicates the location of the interface.

5.2.5. Shock-bubble Interaction

We simulate the interaction of a shock with Mach number 1.22 with a helium bubble immersed in air. This is an established test case to assess the robustness and validity of compressible two-phase numerical methods. Reports are well documented in literature [75, 89, 51] and experimental data are available [90].

A helium bubble with diameter $D = 50\,\text{mm}$ is placed at the origin of the computational domain $x \times y \in [-90\,\text{mm}, 266\,\text{mm}] \times [-44.5\,\text{mm}, 44.5\,\text{mm}]$. The domain is discretized with a grid consisting of 2048×512 cells. The initial shock wave is located 5 mm to the left of the helium bubble. The shock wave travels right and impacts with the helium bubble. We use a CFL number of 0.3 for this case. Figure 17 shows the flow field at two later time instances. The interaction of the initial shock with the helium bubble generates a reflected shock which is travelling to the left and a second shock which is transmitted through the bubble, see Figure 17 on the left. The incident shock wave is visible as a vertical line. The transmitted wave travels faster than the incident shock. The helium bubbles deforms strongly and a re-entrant jet forms. Figure 17 on the right shows the instance in time at which the jet impinges on the interface of the bubble.

The numerical schlieren images and the flow fields in Figure 17 are in good qualitative agreement with results from literature, compare with Figure 7 from [90] or Figure 10 from [51]. Figure 18 shows the temporal evolution of characteristic interface points. The results are in very good quantitative agreement with [89].

6. Single Node Performance

We assess the single node performance of the JAX-Fluids solver on an NVIDIA RTX A6000 GPU. The NVIDIA RTX A6000 GPU provides 48GB of GPU memory and a bandwidth of 768 GB/s. For a performance analysis on a single core of a TPU v3-8 we refer to Appendix B. We additionally provide a roofline analysis of JAX-Fluids kernels in Appendix C.

Here, we conduct simulations of the three-dimensional compressible Taylor-Green vortex (TGV) [91] at $Ma = 0.1$ on a series of grids with increasing resolution. Specifically, we simulate TGVs on 64^3, 128^3, 256^3, and 384^3 cells. The performance analysis is conducted for forward simulations,

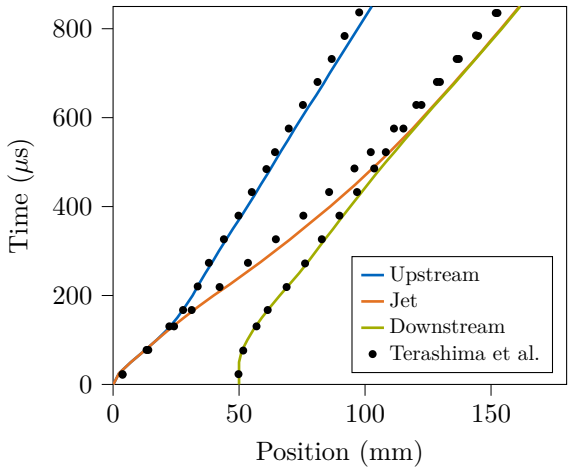


Figure 18: Space-time diagram of three characteristic interface points. Positions of the upstream point (left-most point of the interface), the downstream point (right-most point of the interface), and the jet (left-most point of the interface on the center-line) are tracked. Reference values are taken from Terashima et al. [89].

i.e., without gradient calculation. The two numerical setups under investigation are described in the previous section. We use JAX version 0.3.5 for CUDA 11.5. As JAX-Fluids can handle single- and double-precision computations, we assess the performance for both data types. Table 3 summarizes the results. At 384^3 cells, only the simulation setup *HLLC-float32* did not exceed the memory resources of the A6000 GPU (compare with Table 4). All results reported here are averaged over 5 independent runs. For the *HLLC-float32* setup, JAX-Fluids achieves a peak performance of around 25 ns per cell and per time step (i.e., per degree of freedom). This corresponds to three evaluations of the right-hand side in Equation (9) as we use TVD-RK3 time integration. JAX-Fluids, therefore, provides a strong performance taking into consideration that the code is written entirely in the high-level language Python/JAX. For the *ROE-float32* setup, the computation of the eigenvectors and eigenvalues increases the wall clock time roughly by an order of magnitude. For *HLLC* and *ROE* schemes, we observe that the single-precision calculations are between 2.5 and 3 times faster than the double-precision calculations.

As GPU memory is a critical resource when working with JAX, we investigate the memory consumption of JAX-Fluids. The default behavior of JAX preallocates 90% of GPU memory in order to avoid memory fragmentation. Therefore, to monitor the actual memory consumption, we set `XLA_PYTHON_CLIENT_PREALLOCATE="false"` to disable memory preallocation and force JAX to allocate GPU memory as needed. Additionally, we set `XLA_PYTHON_CLIENT_ALLOCATOR="platform"` which allows JAX to deallocate unused memory. Note that allowing JAX to deallocate unused memory incurs a performance penalty, and we only use this setting to profile the memory consumption. Table 4 summarizes the GPU memory requirements for the aforementioned simulation setups. We refer to the documentation of JAX [27] for more details on GPU memory utilization.

	Mean wall clock time per cell and per time step in $10^{-9}s$				
	32^3	64^3	128^3	256^3	384^3
HLLC-float32	30.09 (6.67)	24.82 (0.11)	25.78 (0.01)	28.69 (0.00)	28.64 (0.00)
HLLC-float64	93.19 (0.58)	95.82 (0.12)	94.42 (0.02)	94.35 (0.00)	-
ROE-float32	230.01 (0.73)	287.92 (0.20)	288.46 (0.08)	287.14 (0.24)	-
ROE-float64	703.40 (1.42)	746.78 (6.17)	759.36 (5.23)	760.09 (6.37)	-

Table 3: Mean wall clock time per cell and per time step. All computations are run on an NVIDIA RTX A6000 GPU. The wall clock times are averaged over five independent runs. Numbers in brackets denote the standard deviation over the five runs.

	Memory pressure				
	32^3	64^3	128^3	256^3	384^3
HLLC-float32	295.6 (1.50)	434.4 (1.50)	1424.4 (1.50)	9141.6 (1.50)	29849.2 (2.04)
HLLC-float64	353.6 (2.33)	623.6 (2.33)	2631.6 (2.33)	18275.6 (2.33)	-
ROE-float32	626.0 (1.79)	818.8 (2.04)	2255.2 (2.04)	13546.0 (1.79)	-
ROE-float64	688.0 (2.83)	1068.4 (2.33)	3938.0 (2.83)	26504.4 (2.33)	-

Table 4: GPU Memory Pressure in megabytes (MB). The memory consumption is averaged over five independent runs. Numbers in brackets denote the standard deviation over the five runs.

7. Machine Learning in JAX-Fluids

Having showcased that JAX-Fluids functions very well as a modern and easy-to-use fluid dynamics simulator, we now discuss its capabilities for ML research, in particular its automatic differentiation capabilities for end-to-end optimization. In this section, we demonstrate that we can successfully differentiate through the entire JAX-Fluids solver. We validate the AD gradients for single- and two-phase flows. We then showcase the potential of JAX-Fluids by training a data-driven Riemann solver leveraging end-to-end optimization.

7.1. Deep Learning Fundamentals

Given a data set of input-outputs pairs $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_N, \mathbf{y}_N)\}$ with $\mathbf{x} \in \mathcal{X}$ and $\mathbf{y} \in \mathcal{Y}$, supervised learning tries to find a function $f : \mathcal{X} \to \mathcal{Y}$ which (approximately) minimizes an average loss

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{y}_i, f(\mathbf{x}_i)), \tag{38}$$

where $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a suitable loss function. $\mathcal{X}$ and $\mathcal{Y}$ are input and output spaces, and $f \in \mathcal{F}$, where $\mathcal{F}$ is the hypothesis space. We use $\hat{\mathbf{y}}_i$ to denote the output of the function f for input $\mathbf{x}_i$,

$\hat{\mathbf{y}}_i = f(\mathbf{x}_i)$. A popular loss in regression tasks is the mean-squared error (MSE)

$$\mathcal{L} = MSE(\mathbf{x}, \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2. \tag{39}$$

One possible and highly-expressive class of functions are deep neural networks (DNNs) [22, 92]. DNNs are parameterizable nonlinear compound functions, $f = f_\theta$, where the network parameters consist of weights and biases, $\theta = \{\mathbf{W}, \mathbf{b}\}$. DNNs consist of multiple hidden layers of units between input layer and output layer. The values in each layer are called activations. Multilayer perceptrons (MLPs) are one particular kind of DNN in which adjacent layers are densely connected [92]. In MLPs, we compute the activations $\mathbf{a}^l$ in layer l from the activations of the previous layer $\mathbf{a}^{l-1}$,

$$\mathbf{a}^l = \sigma(\mathbf{W}^{l-1}\mathbf{a}^{l-1} + \mathbf{b}^{l-1}). \tag{40}$$

Here, $\mathbf{W}^{l-1}$ is the weight matrix linking layers $l-1$ and l, $\mathbf{b}^{l-1}$ is the bias vector, and $\sigma(\cdot)$ is the element-wise nonlinearity. Typically, DNNs are trained by minimizing $\mathcal{L}$ via mini-batch gradient descent or more advanced optimization routines like AdaGrad [93] or Adam [80].

7.2. Optimization through PDE Trajectories

Machine learning has the potential to discover and learn novel data-driven numerical algorithms for the numerical computation of fluid dynamics. For supervised learning, we need input-ouput pairs. In fluid dynamics, exact solutions rarely exist for complex flow. We usually take highly-resolved numerical simulations as exact.

In the context of differentiable physics, end-to-end optimization usually refers to supervised learning of ML models which receive gradients that are backpropagated through a differentiable physics simulator. Here, the ML model is placed inside a differentiable PDE solver. A trajectory obtained from a forward simulation of the PDE solver is compared to a ground truth trajectory, and the derivatives of the loss are propagated across the temporal sequence, i.e., the trajectory. We denote a trajectory of states as

$$\tau = \left\{\mathbf{U}^1, ..., \mathbf{U}^{N_T}\right\}. \tag{41}$$

The differentiable solver, e.g., JAX-Fluids, can be interpreted as a parameterizable generator $\mathcal{G}_\theta$ of such a trajectory starting from the initial condition $\mathbf{U}_0$.

$$\tau_\theta^{PDE} = \left\{\mathbf{U}^1, ..., \mathbf{U}^{N_T}\right\} = \mathcal{G}_\theta(\mathbf{U}_0) \tag{42}$$

Our loss objective is the difference between the trajectory τ_θ^{PDE} and a ground truth trajectory $\hat{\tau} = \left\{\hat{\mathbf{U}}^1, ..., \hat{\mathbf{U}}^{N_T}\right\}$. For example, using the MSE in state space

$$\mathcal{L}^\tau = \frac{1}{N_T}\sum_{i=1}^{N_T} MSE(\mathbf{U}^i, \hat{\mathbf{U}}^i). \tag{43}$$

The derivatives of the loss with respect to the tuneable parameters $\partial\mathcal{L}/\partial\theta$ are backpropagated across the simulation trajectory and through the entire differentiable PDE solver. The ML model is then optimized by using $\partial\mathcal{L}/\partial\theta$ in a gradient-based optimization routine.

In particular, multiple steps through the simulator can be chained together. Thereby, the ML model observes the full dynamics of the underlying PDE and learns how its actions influence the entire simulation trajectory. Naturally, the trained model is equation-specific and physics-informed. This training procedure alleviates the problem of distribution mismatch between training and test data as the model sees its own outputs during the training phase. Additionally, the ML model could potentially account for approximation errors of other parts of the solver.

JAX-Fluids allows us to take gradients through an entire CFD simulation trajectory by applying the JAX operation `jax.grad` to any scalar observable of the state trajectory. We want to stress that JAX-Fluids thereby differentiates for each time step through complex subfunctions such as the spatial reconstruction, Riemann solvers, or two-phase interactions.

7.3. Validation of Automatic Differentiation Gradients

Before we showcase the full potential of JAX-Fluids for learning data-driven numerical schemes by end-to-end optimization, we first validate the gradients obtained from automatic differentiation by comparing them with gradients obtained from finite-differences. We consider a single shock wave with shock Mach number M_S propagating into a fluid at rest. We fix the state ahead of the shock with $\rho_R = p_R = 1$ and $u_R = 0$. The Rankine-Hugoniot relations [54] determine the state behind the shock wave as a function of the shock Mach number M_S. As the shock wave crosses the computational domain, the integral entropy increases. The described setup is depicted in Figure 19 on left. The left and right states are separated by an initial shock discontinuity. We consider a single-phase and two-phase setup. In both setups all fluids are modeled by the ideal gas law. In the former, the same fluid is left and right of the initial shock wave and $\gamma_L = \gamma_R$. In the latter, two immiscible fluids are separated by the shock wave, i.e., $\gamma_L \neq \gamma_R$. For the second setup, we make use of the entire level-set algorithm as described earlier.

As the shock wave propagates into the domain, the integral entropy in the domain increases by ΔS, see the schematic in the middle of Figure 19. The integral entropy at time t is defined by

$$S(t) = \int_\Omega \rho(x,t)s(x,t)dx, \tag{44}$$

and the increase in integral entropy is

$$\Delta S(t) = S(t) - S(t_0) = \int_\Omega \left(\rho(x,t)s(x,t) - \rho(x,t=0)s(x,t=0)\right) dx. \tag{45}$$

In the simplified setting under investigation, the increase in integral entropy at any fixed point in time, say t^n, is solely determined by the shock Mach number M_S, i.e., $\Delta S^n = \Delta S(t = t^n) = \Delta S(M_S)$. We

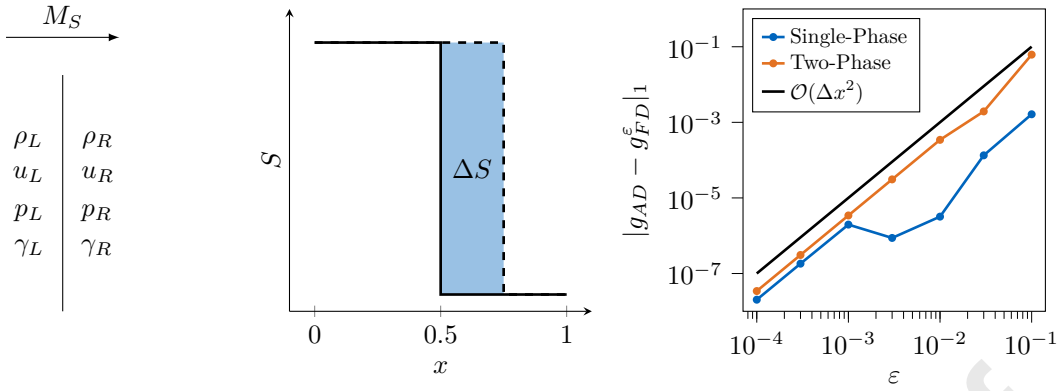


Figure 19: Left: Schematic of the computational setup. The constant initial states are separated by a single right-running shock discontinuity. Middle: Schematic of the total entropy increase. Right: Error convergence of the gradient obtained by second-order central finite-difference approximation with respect to the gradient obtained by automatic differentiation, i.e., $|g_{AD} - g^{\varepsilon}_{FD}|_p$.

let the shock wave propagate for n steps with fixed Δt, i.e., $t^n = n\Delta t$, and calculate the gradient of the total entropy increase ΔS^n with respect to the shock Mach number M_S.

$$g = \frac{\partial \Delta S^n}{\partial M_S} \tag{46}$$

We compute the gradient with automatic differentiation g_{AD} and with second-order central finite-differences according to

$$g^{\varepsilon}_{FD} = \frac{\Delta S^n(M_S + \varepsilon) - \Delta S^n(M_S - \varepsilon)}{2\varepsilon}. \tag{47}$$

Here, we set $M_S = 2$. We use the *HLLC* setup described in the previous section. We set $\Delta t = 1 \times 10^{-2}$ and $n = 5$. In the single-phase case $\gamma_L = \gamma_R = 1.4$, in the two-phase case $\gamma_L = 1.4$ and $\gamma_R = 1.667$. On the right of Figure 19, we visualize the l_1 norm between the gradients g_{AD} and g^{ε}_{FD} for the single-phase and two-phase setup. We choose $\varepsilon \in [1e-1, 3e-2, 1e-2, 3e-3, 1e-3, 3e-4, 1e-4]$. We observe that the finite-difference approximations converge with second-order to the respective automatic differentiation gradients. We conclude that automatic differentiation through the entire JAX-Fluids code works and gives correct gradients. In passing, we note that although the described setting is a simplified one-dimensional setup, and we only differentiate with respect to a single parameter, the AD call has to backpropagate through multiple integration steps with several Runge-Kutta substeps each accompanied by calls to spatial reconstruction and Riemann solver. Especially in the two-phase setup, gradients are obtained through the entire level-set computational routines including level-set advection, level-set reinitilization, and the extension procedure.

7.4. End-to-end Optimization of a Riemann Solver

Automatic differentiation yields the opportunity to optimize and learn numerical schemes from data by end-to-end optimization through a numerical simulator [41, 45, 46]. In this section, we want

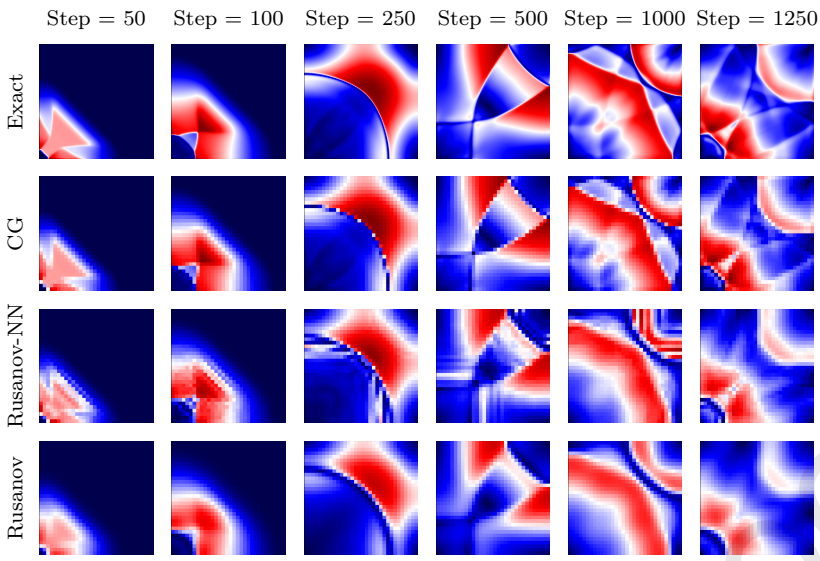


Figure 20: Trajectories of the absolute velocity. From top to bottom: Ground truth (Exact) on 128×128, Coarse-grained (CG) on 32×32, Rusanov-NN on 32×32, and Rusanov on 32×32. For each time step, values are normalized with the minimum and maximum value of the exact solution.

to showcase how JAX-Fluids is able to learn a numerical flux function (i.e., an approximate Riemann solver) by minimizing a loss between predicted trajectory and ground truth trajectory. We optimize the popular Rusanov flux function (also known as local Lax-Friedrichs flux function). The Rusanov flux at the cell face $x_{i+1/2}$ is

$$\mathbf{F}^{\text{Rusanov}}_{i+1/2} = \frac{1}{2}(\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2}\alpha(\mathbf{U}_R - \mathbf{U}_L). \tag{48}$$

Here, $\mathbf{U}_{L/R}$ and $\mathbf{F}_{L/R}$ are the left and right sided cell face reconstructions of the conservative variable and the flux. α is the scalar numerical viscosity. For the classical Rusanov method the numerical viscosity is defined at each cell face $\alpha_{\text{Rusanov}} = \max\{|u_L - c_L|, |u_L + c_L|, |u_R - c_R|, |u_R + c_R|\}$. u is the cell face normal velocity and c is the local speed of sound. It is well known that although the Rusanov method yields a stable solution, the excess numerical diffusion leads to smearing out of the solution. As a simple demonstration of the AD-capabilities of our CFD solver, we introduce the Rusanov-NN flux with the dissipation $\alpha^{\text{NN}}_{\text{Rusanov}} = NN(|\Delta u|, u_M, c_M, |\Delta s|)$ to be optimized. The dissipation is output of a multi-layer perceptron which takes as inputs the jump in normal velocity $\Delta u = |u_R - u_L|$, the mean normal velocity $u_M = \frac{1}{2}(u_L + u_R)$, the mean speed of sound $c_M = \frac{1}{2}(c_L + c_R)$, and the entropy jump $\Delta s = |s_R - s_L|$. The network is composed of two hidden layers with 32 nodes each and an output layer with a single node. We use RELU activations for the hidden layers. An exponential activation function in the output layer guarantees $\alpha^{\text{NN}}_{\text{Rusanov}} \geq 0$. We set up a highly-resolved simulation of a two-dimensional implosion test case to generate the ground truth trajectory.

The initial conditions are a diagonally placed jump in pressure and density,

$$(\rho, u, v, p) = \begin{cases} (0.14, 0, 0, 0.125) & \text{if } x + y \leq 0.15, \\ (1, 0, 0, 1) & \text{if } x + y > 0.15, \end{cases} \tag{49}$$

on a domain with extent $x \times y \in [0, 1] \times [0, 1]$. A shock, a contact discontinuity, and a rarefaction wave emanate from the initial discontinuity and travel along the diagonal. The shock is propagating towards the lower left corner and is reflected by the walls resulting in a double Mach reflection, while the rarefaction wave is travelling into the open domain. The high resolution simulation is run on a mesh with 128×128 cells with a WENO3-JS cell face reconstruction, TVD-RK2 integration scheme, and the HLLC Riemann solver. We use a fixed time step $\Delta t = 2.5 \times 10^{-4}$ and sample the trajectory every $\Delta t_{\text{CG}} = 10^{-3}$, which is the time step for the coarse-grained trajectories. A trajectory of 2501 time steps is generated, i.e., $t \in [0, 2500\Delta t_{\text{CG}}]$. Exemplary time snapshots for the absolute velocity are visualized in the top row of Figure 20. We obtain the ground truth data after coarse-graining the high-resolution trajectory onto 32×32 points, see second row of Figure 20.

The dissipation network model is trained in a supervised fashion by minimizing the loss between the coarse-grained (CG) trajectory and the simulation produced by the Rusanov-NN model. The loss function is defined as the mean-squared error between the predicted and coarse-grained primitive state vectors, $\mathbf{W}^{\text{NN}}$ and $\mathbf{W}^{\text{CG}}$, over a trajectory of length N_T,

$$L = \frac{1}{N_T} \sum_{i=1}^{N_T} MSE(\mathbf{W}_i^{\text{NN}}, \mathbf{W}_i^{\text{CG}}). \tag{50}$$

The training data set consists of the first 1000 time steps of the coarse grained reference solution. During training, the model is unrolled for $N_T = 15$ time steps. We use the Adam optimizer with a constant learning rate $5e-4$ and a batch size of 20. The Rusanov-NN model is trained for 100 epochs. Although we have trained on trajectories of length 15, the trained model is then evaluated for the entire trajectory of 2501 time steps. Figure 20 compares the results of the Rusanov and the learned Rusanov-NN flux functions. The NN-Rusanov flux is less dissipative than the classical Rusanov scheme and recovers small scale flow structures very well, e.g., see time step 100 in Figure 20. The NN-Rusanov flux stays stable over the course of the simulation and consistently outperforms the Rusanov flux, see the relative errors in pressure and density in Figure 21. The ML model even performs very well outside the training set (time steps larger than 1000).

8. Conclusion

We have presented JAX-Fluids, a comprehensive state-of-the-art fully-differentiable python package for compressible three-dimensional computational fluid dynamics. Machine learning is becoming more and more dominant in the physical and engineering sciences. Especially, fluid dynamics represents a field in which ML techniques and data-driven methods show very promising results and seem to

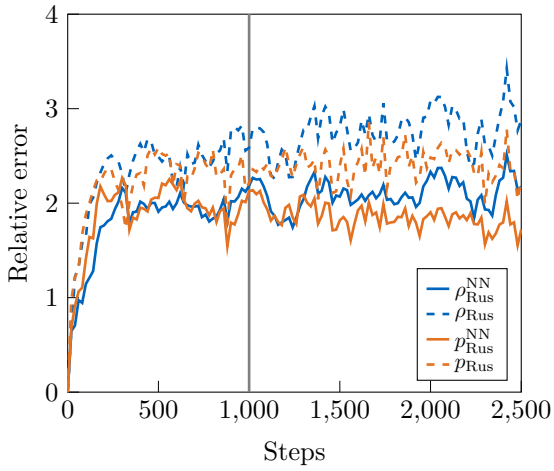


Figure 21: Relative l_1 error for density (blue) and pressure (orange). The gray line indicates the training horizon.

have a high potential. Despite the recent surge of ML-CFD research, a comprehensive state-of-the-art differentiable CFD solver has not been published. JAX-Fluids provides researchers at the intersection of fluid dynamics and deep learning the opportunity to explore new data-driven numerical models for fluid dynamics. JAX-Fluids offers powerful high-order numerical methods for a wide variety of fluid dynamics problems, e.g., turbulent flows, flows with arbitrary solid boundaries, compressible flows, and two-phase flows. The modular architecture of JAX-Fluids makes integration of custom submodules easy and straightforward.

Although JAX-Fluids covers a wide range of flow physics, some intriguing and complex phenomena like combustion, fluid-structure interaction, or cavitation cannot yet be modeled with JAX-Fluids. For the future, we plan to implement appropriate numerical methods.

Currently, by far the largest limitation of JAX-Fluids is the available memory of the GPU. Although JAX-Fluids scales out of the box to problem sizes with roughly 400 million degrees of freedom (DOFs) on a single modern GPU, many problems in fluid dynamics require higher DOFs. Generally, there are two ways of tackling this problem: adaptive multiresolution [94, 95] and parallelization to multiple GPUs, e.g., [96, 2]. Adaptive multiresolution increases the mesh resolution in areas of interest (e.g., where small scale structures are present) while using a much coarser resolution in other parts of the flow field. Compared to a uniform mesh, adaptive multiresolution increases the efficiency of the simulation in terms of computational resources. To the knowledge of the authors, multiresolution strategies seem to be problematic with a jit-compiled code framework such as JAX, as the computegraph has to be static. A second approach for increasing the computable problem size is parallelization. The latest version of JAX [27] as well as other works [97] propose different parallelization strategies for JAX algorithms. We are currently pursuing the aforementioned approaches to solve the present memory limitations and will incorporate parallelization extensions into future releases of JAX-Fluids.

	Mean wall clock time per cell and per time step in $10^{-9}s$			
	32^3	64^3	128^3	256^3
HLLC-float32	305.14 (0.83)	100.05 (0.16)	49.60 (0.01)	35.23 (0.00)
HLLC-float64	1468.8 (1.00)	596.20 (0.11)	373.76 (0.00)	-

Table B.5: Mean wall clock time per cell and per time step on a single core of a TPU v3-8. The wall clock times are averaged over five independent runs. Numbers in brackets denote the standard deviation over the five runs.

Acknowledgements

Appendix A. Numerical and Case Setup Files for Sod Shock Tube Test Case

Figure A.22 and Figure A.23 show exemplary numerical setup and case setup files for JAX-Fluids.

Appendix B. Running JAX-Fluids on a TPU v3-8

To demonstrate that JAX-Fluids runs on TPUs, we perform scaling tests on a single core of a TPU v3-8 using the Google Cloud service. A single core of this particular TPU has 16GB memory. Analogously to the scaling runs on a GPU (compare Section 6), we benchmark the mean wall clock time per cell and per time step for the Taylor-Green-Vortex [91] using the *HLLC* setup. We simulate the resolution 32^3, 64^3, 128^3, and 256^3. At 256^3 cells, only the simulation setup *HLLC-float32* did not exceed the memory resources of the TPU v3-8 (single core).

Table B.5 illustrates the mean wall clock time per cell and per time step in nanoseconds. For the *HLLC-float32* setup JAX-Fluids achieves a peak performance of around 35 ns per cell and per time step, provided the TPU is fully saturated. The *HLLC-float64* setup achieves a peak performance of around 373 ns. Switching from single- to double-precision results in roughly 10 times higher wall clock times on a TPU, while the factor was only about 3 for GPUs (compare with Table 3). Workloads that require high-precision arithmetic are not suitable for TPUs. TPUs only support 64bit floating point operations at the software level [98]. This is reflected by the significant decrease in performance from single- to double-precision.

Appendix C. Roofline Analysis

We complement the time-per-cell numbers from Section 6 with a roofline analysis [99]. The roofline model visualizes the performance of a compute kernel and indicates potential bottlenecks but also potentials for performance optimization. The performance of the compute kernel is visualized as a function of machine peak performance, machine peak bandwidth, and the arithmetic intensity. The arithmetic intensity is characteristic of the compute kernel and describes the ratio of total floating

```
{
    "conservatives": {
        "halo_cells": 4,
        "time_integration" : {
            "integrator": "RK3",
            "CFL": 0.9
        },
        "convective_fluxes": {
            "convective_solver": "FLUX-SPLITTING",
            "flux_splitting": "ROE",
            "signal_speed": "EINFELDT",
            "spatial_reconstructor": "WENO5-JS",
            "is_safe_reconstruction": false,
            "reconstruction_var": "CHAR-CONSERVATIVE"
        },
        "dissipative_fluxes": {
            "reconstruction_stencil": "R4",
            "derivative_stencil_center": "DC4",
            "derivative_stencil_face": "DF4"
        }
    },
    "active_physics": {
        "is_convective_flux": true,
        "is_viscous_flux": false,
        "is_heat_flux": false,
        "is_volume_force": false
    },
    "active_forcings": {
        "is_mass_flow_forcing": false,
        "is_temperature_forcing": false,
        "is_turb_hit_forcing": false
    },
    "output":  {
        "is_double_precision_compute": false,
        "is_double_precision_output": false,
        "derivative_stencil": "DC4",
        "quantities": {
            "primes": ["density", "velocityX", "pressure", "temperature"]
        }
    }
}
```

Figure A.22: Numerical setup *json* file for the Sod shock tube test case.

```
{
    "general": {
        "case_name": "sod",
        "save_path": "./results",
        "end_time": 0.2,
        "save_dt": 0.01
    },
    "domain": {
        "x": {
            "cells": 100,
            "range": [0.0, 1.0]
        },
        "y": {
            "cells": 1,
            "range": [0.0, 1.0]
        },
        "z": {
            "cells": 1,
            "range": [0.0, 1.0]
        }
    },
    "boundary_condition": {
        "types": {
            "east": "neumann",
            "west": "neumann",
            "north": "inactive",
            "south": "inactive",
            "top": "inactive",
            "bottom": "inactive"
        }
    },
    "initial_condition": {
        "rho": "lambda x: 1.0*(x <= 0.5) + 0.125*(x > 0.5)",
        "u": 0.0,
        "v": 0.0,
        "w": 0.0,
        "p": "lambda x: 1.0*(x <= 0.5) + 0.1*(x > 0.5)"
    },
    "material_properties": {
        "type": "IdealGas",
        "dynamic_viscosity": 0.0,
        "bulk_viscosity": 0.0,
        "thermal_conductivity": 0.0,
        "specific_heat_ratio": 1.4,
        "specific_gas_constant": 1.0
    },
    "nondimensionalization_parameters": {
        "density_reference": 1.0,
        "length_reference": 1.0,
        "velocity_reference": 1.0,
        "temperature_reference": 1.0
    }
}
```

Figure A.23: Case setup *json* file for the Sod shock tube test case.

point operations to the total amount of data transferred. The roofline model gives an upper bound for the performance (in floating point operations per second, i.e, FLOP/s),

$$\text{GFLOP/s} \leq \min \begin{cases} \text{Peak GFLOP/s}, \\ \text{Peak GB/s} \times \text{Arithmetic Intensity}. \end{cases} \tag{C.1}$$

We apply the roofline analysis to the Dynamic Random Access Memory (DRAM). I.e., peak bandwidth and data transfer are measured with respect to DRAM.

As described in Section 4, the `do_integration_step` method is one of the functions on the highest level of the code which is just-in-time-compiled. `do_integration_step` integrates the entire flow field for one time step and is the most compute-intensive part of the JAX-Fluids code. We apply the roofline analysis to `do_integration_step` to obtain a detailed performance analysis. We test the *HLLC* and *ROE* setup for single- and double-precision and different spatial resolutions, as introduced in Section 5.

When jit-compiling a function, JAX invokes the XLA compiler. The XLA-compiler generates optimized kernels, especially by fusion of low-level operations into so-called *fused kernels*. When we jit-compile a high-level function such as `do_integration_step` which is composed of a series of complex subroutines, the XLA-compiler generates dozens of fused kernels which together make up the high-level function call. We use the NVIDIA Nsight Compute Command Line Interface to profile each of these kernels according to [100]. For every kernel, we measure the temporal duration, the number of floating point operations, and the amount of data transfer. We can then compute a time-averaged performance metric over the fused kernels to provide a single metric for the top-level function that we want to profile.

We conduct all experiments on an NVIDIA RTX A6000 GPU. The A6000 GPU has a nominal bandwidth of 768 GB/s. The nominal single-precision performance is 38.7 TFLOP/s and the nominal double-precision performance is 1.2 TFLOP/s.

In Figure C.24, we show the performance of the `do_integration_step` routine for aforementioned *HLLC* and *ROE* setups at different spatial resolutions. In all tests, we use TVD-RK3 time integration and WENO5-JS reconstruction. We distinguish between single- and double-precision performance. There are multiple takeaways from this analysis. First, we observe that the arithmetic intensity decreases as we begin to increase the spatial resolution and converges for resolutions of 128^3 and higher. This is due to the fact that we measure the arithmetic intensity with respect to DRAM. At low resolutions the problem size is small enough, such that L1 and L2 caches can be used more extensively (not shown) while data transfer from and to DRAM is relatively low. This results in higher DRAM arithmetic intensities. When increasing the problem size, the DRAM arithmetic intensities converge to a single value for each numerical configuration. Second, the percentage of attainable peak performance increases as we increase the spatial resolution. On the right of Figure C.24, we plot

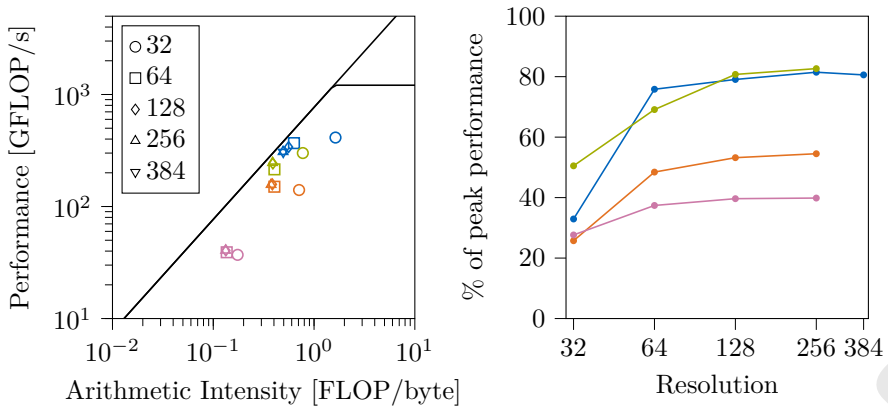


Figure C.24: Roofline analysis (left) and percentage of attainable peak performance (right) for four different numerical setups: *HLLC-float32* (blue), *HLLC-float64* (orange), *ROE-float32* (green), and *ROE-float64* (pink). Arithmetic intensity is measured with respect to to DRAM.

the ratio of achieved to peak performance. We can clearly see that the values converge for different numerical setups when increasing the problem size. The *HLLC-float32* setup reaches around 81% of peak performance while *ROE-float32* reaches around 83%. The double-precision setups achieve lower values. While *HLLC-float64* levels out at roughly 55% of the peak performance, *ROE-float64* only achieves 40%. Third, we can clearly see that all kernels are in the memory bound regime. This is in line with results from literature, as CFD - especially stencil-based finite-difference and finite-volume schemes - typically are memory bound, e.g., [101].

We also conduct a roofline analysis for different spatial reconstruction schemes at fixed spatial resolution. We vary the spatial reconstruction scheme from a first-order reconstruction to a seventh-order WENO scheme. The spatial resolution is fixed at 128^3. Figure C.25 shows the roofline analysis for *HLLC* and *ROE* setups. In the left part of Figure C.25, we observe that an increase of the spatial order generally increases the arithmetic intensity. Only for the *ROE-float64* setup, the arithmetic intensity remains nearly constant. The peak performance for *HLLC-float32* stays roughly constant at around 81% of peak performance for different reconstruction schemes. For *HLLC-float64*, the peak performance decreases from 73% for a first-order scheme to 40% for WENO7-JS. For both, *ROE-float32* and *ROE-float64*, the achievable performance increases with higher order reconstruction schemes. *ROE-float32* with WENO7-JS uses 82% of the attainable peak performance.

Data Availability Statement

JAX-Fluids is available under the GNU GPLv3 license at `https://github.com/tumaer/JAXFLUIDS`.

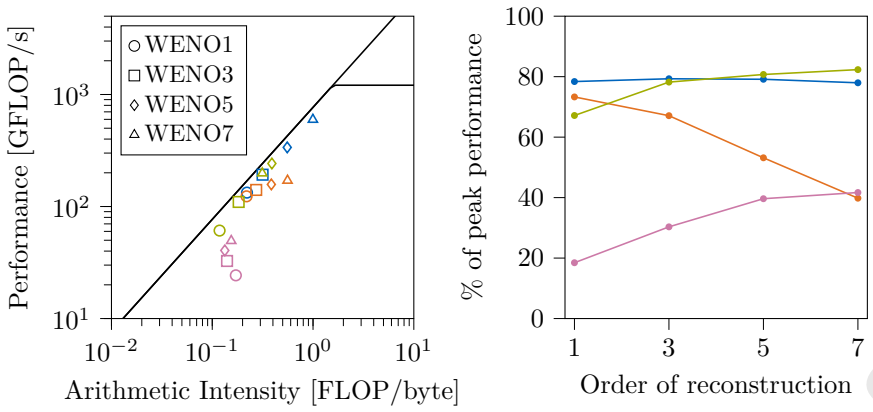


Figure C.25: Roofline analysis (left) and percentage of attainable peak performance (right) for four different numerical setups: *HLLC-float32* (blue), *HLLC-float64* (orange), *ROE-float32* (green), and *ROE-float64* (pink). Arithmetic intensity is measured with respect to to DRAM.

References

[1] P. Bauer, A. Thorpe, G. Brunet, Nature 525 (7567) (2015) 47–55. doi:10.1038/nature14956.

[2] D. Häfner, R. Nuterman, M. Jochum, Journal of Advances in Modeling Earth Systems 13 (12) (2021). doi:10.1029/2021MS002717.
URL https://onlinelibrary.wiley.com/doi/10.1029/2021MS002717

[3] N. Nowak, P. P. Kakade, A. V. Annapragada, Annals of Biomedical Engineering 31 (4) (2003) 374–390. doi:10.1114/1.1560632.

[4] B. M. Johnston, P. R. Johnston, S. Corney, D. Kilpatrick, Journal of Biomechanics 37 (5) (2004) 709–720. doi:10.1016/j.jbiomech.2003.09.016.

[5] J. D. Denton, W. N. Dawes, Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering 213 (2) (1999) 107–124. doi:10.1243/0954406991522211.
URL http://journals.sagepub.com/doi/10.1243/0954406991522211

[6] M. O. Hansen, J. N. Sørensen, S. Voutsinas, N. Sørensen, H. A. Madsen, Progress in Aerospace Sciences 42 (4) (2006) 285–330. doi:10.1016/j.paerosci.2006.10.002.
URL https://linkinghub.elsevier.com/retrieve/pii/S0376042106000649

[7] B. Sanderse, S. P. Van Der Pijl, B. Koren, Wind Energy 14 (7) (2011) 799–819. doi:10.1002/we.458.
URL https://onlinelibrary.wiley.com/doi/10.1002/we.458

[8] Z. Lyu, Z. Xu, J. Martins, The Eighth International Conference on Computational Fluid Dynamics (2014) 18.
URL https://info.aiaa.org/tac/ASG/APATC/AeroDesignOpt-DG/default.aspx

[9] K. Duraisamy, G. Iaccarino, H. Xiao, Annual Review of Fluid Mechanics 51 (1) (2019) 357–377. doi:10.1146/annurev-fluid-010518-040547.
URL https://www.annualreviews.org/doi/10.1146/annurev-fluid-010518-040547

[10] M. P. Brenner, J. D. Eldredge, J. B. Freund, Physical Review Fluids 4 (10) (2019) 100501. doi:10.1103/PhysRevFluids.4.100501.
URL https://link.aps.org/doi/10.1103/PhysRevFluids.4.100501

[11] S. L. Brunton, B. R. Noack, P. Koumoutsakos, Annual Review of Fluid Mechanics 52 (1) (2020) 477–508. doi:10.1146/annurev-fluid-010719-060214.

[12] A. Günes Baydin, B. A. Pearlmutter, A. Andreyevich Radul, J. Mark Siskind, Journal of Machine Learning Research 18 (2018) 1–43. arXiv:1502.05767.

[13] F. H. Harlow, Journal of Computational Physics 195 (2) (2004) 414–433. doi:10.1016/j.jcp.2003.09.031.

[14] B. van Leer, Journal of Computational Physics 32 (1) (1979) 101–136. doi:10.1016/0021-9991(79)90145-1.

[15] P. L. Roe, Journal of Computational Physics 43 (2) (1981) 357–372. doi:10.1016/0021-9991(81)90128-5.

[16] P. Woodward, P. Colella, Journal of Computational Physics 54 (1) (1984) 115–173. doi:10.1016/0021-9991(84)90142-6.
URL https://www.sciencedirect.com/science/article/pii/0021999184901426

[17] E. F. Toro, M. Spruce, W. Speares, Shock waves 4 (1) (1994) 25–34.
URL http://link.springer.com/article/10.1007/BF01414629https://link.springer.com/content/pdf/10.1007%2FBF01414629.pdf

[18] M. S. Liou, Journal of Computational Physics 129 (2) (1996) 364–382. doi:10.1006/jcph.1996.0256.

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, Neural Computation 1 (4) (1989) 541–551. doi:10.1162/neco.1989.1.4.541.

[20] S. Hochreiter, J. Schmidhuber, Neural Computation 9 (8) (1997) 1735–1780. doi:10.1162/neco.1997.9.8.1735.

[21] Y. LeCun, K. Kavukcuoglu, C. Farabet, in: ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems, 2010, pp. 253–256. doi:10.1109/ISCAS.2010.5537907.

[22] Y. LeCun, Y. Bengio, G. Hinton, Nature 521 (7553) (2015) 436–444. doi:10.1038/nature14539.
URL http://www.nature.com/articles/nature14539

[23] B. M. Lake, R. Salakhutdinov, J. B. Tenenbaum, Science 350 (6266) (2015) 1332–1338. doi:10.1126/science.aab3050.

[24] B. Alipanahi, A. Delong, M. T. Weirauch, B. J. Frey, Nature Biotechnology 33 (8) (2015) 831–838. doi:10.1038/nbt.3300.
URL http://www.nature.com/articles/nbt.3300

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, G. Brain, Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (2016).
URL https://tensorflow.org.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, in: Advances in Neural Information Processing Systems, Vol. 32, 2019. arXiv:1912.01703.

[27] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang (2018).

[28] S. L. Brunton, J. L. Proctor, J. N. Kutz, W. Bialek, Proceedings of the National Academy of Sciences of the United States of America 113 (15) (2016) 3932–3937. doi:10.1073/pnas.1517384113.

[29] M. Raissi, P. Perdikaris, G. E. Karniadakis, Journal of Computational Physics 378 (2019) 686–707. doi:10.1016/j.jcp.2018.10.045.

[30] A. B. Buhendwa, S. Adami, N. A. Adams, Machine Learning with Applications 4 (2021) 100029. doi:10.1016/j.mlwa.2021.100029.
URL https://doi.org/10.1016/j.mlwa.2021.100029

[31] S. L. Brunton, M. S. Hemati, K. Taira, Theoretical and Computational Fluid Dynamics 34 (4) (2020) 333–337. doi:10.1007/s00162-020-00542-y.

[32] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Nature Reviews Physics 3 (6) (2021) 422–440. doi:10.1038/s42254-021-00314-5.

[33] A. Beck, D. Flad, C. D. Munz, Journal of Computational Physics 398 (2019). doi:10.1016/j.jcp.2019.108910.

[34] H. V. Patel, A. Panda, J. A. Kuipers, E. A. Peters, Computers and Fluids 193 (2019). doi:10.1016/j.compfluid.2019.104263.

[35] A. B. Buhendwa, D. A. Bezgin, N. A. Adams, Journal of Computational Physics 457 (2022). doi:10.1016/j.jcp.2022.111049.

[36] B. Stevens, T. Colonius, Theoretical and Computational Fluid Dynamics 34 (4) (2020) 483–496. doi:10.1007/s00162-020-00531-1.
URL http://arxiv.org/abs/2002.02521

[37] D. A. Bezgin, S. J. Schmidt, N. A. Adams, Journal of Computational Physics 452 (2022) 110920. doi:10.1016/j.jcp.2021.110920.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999121008159

[38] A. Jameson, Journal of Scientific Computing 3 (3) (1988) 233–260. doi:10.1007/BF01061285.

[39] J. Sirignano, J. F. MacArt, J. B. Freund, Journal of Computational Physics 423 (2020). arXiv:1911.09145, doi:10.1016/j.jcp.2020.109811.

[40] C. A. Ströfer, H. Xiao, Theoretical and Applied Mechanics Letters 11 (4) (2021). arXiv:2104.04821, doi:10.1016/j.taml.2021.100280.

[41] Y. Bar-Sinai, S. Hoyer, J. Hickey, M. P. Brenner, Proceedings of the National Academy of Sciences of the United States of America 116 (31) (2019) 15344–15349. doi:10.1073/pnas.1814058116.

[42] J. Zhuang, D. Kochkov, Y. Bar-Sinai, M. P. Brenner, S. Hoyer, Physical Review Fluids 6 (6) (2021). doi:10.1103/PhysRevFluids.6.064605.

[43] O. Owoyele, P. Pal, Energy and AI 7 (2022). doi:10.1016/j.egyai.2021.100118.

[44] K. Um, R. Brand, Y. Fei, P. Holl, N. Thuerey, in: Advances in Neural Information Processing Systems, Vol. 2020-Decem, 2020. arXiv:2007.00016.
URL https://github.com/tum-pbs/Solver-in-the-Loop.

[45] D. A. Bezgin, S. J. Schmidt, N. A. Adams, Journal of Computational Physics 437 (2021) 110324. doi:10.1016/j.jcp.2021.110324.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999121002199

[46] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, S. Hoyer, Proceedings of the National Academy of Sciences 118 (21) (2021). doi:10.1073/pnas.2101784118.
URL https://www.pnas.org/content/118/21/e2101784118

[47] S. S. Schoenholz, E. D. Cubuk, in: Advances in Neural Information Processing Systems, Vol. 2020-Decem, 2020. arXiv:1912.04232.
URL www.github.com/google/jax-md.

[48] V. Coralic, T. Colonius, Journal of Computational Physics 274 (2014) 95–121. doi:10.1016/j.jcp.2014.06.003.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999114004082

[49] G. S. Jiang, C. W. Shu, Journal of Computational Physics 126 (1) (1996) 202–228. doi:10.1006/jcph.1996.0130.

[50] S. Gottlieb, C.-W. Shu, Mathematics of computation of the American Mathematical Society 67 (221) (1998) 73–85.
URL http://www.ams.org/mcom/1998-67-221/S0025-5718-98-00913-2/

[51] N. Hoppe, J. M. Winter, S. Adami, N. A. Adams, Computer Physics Communications 272 (2022) 108246. doi:10.1016/j.cpc.2021.108246.

[52] J. Qiu, C. W. Shu, Journal of Computational Physics 183 (1) (2002) 187–209. doi:10.1006/jcph.2002.7191.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999102971913

[53] A. Harten, P. D. Lax, B. van Leer, SIAM Review 25 (1) (1983) 35–61. doi:10.1137/1025002.
URL http://epubs.siam.org/doi/10.1137/1025002

[54] E. F. Toro, Riemann solvers and numerical methods for fluid dynamics a practical introduction, 3rd Edition, Springer Verlag, 2009.

[55] S. Osher, J. A. Sethian, Journal of Computational Physics 79 (1) (1988) 12–49. doi:10.1016/0021-9991(88)90002-2.

[56] X. Y. Hu, B. C. Khoo, N. A. Adams, F. L. Huang, J. Comput. Phys. 219 (2) (2006) 553–578. doi:10.1016/j.jcp.2006.04.001.
URL http://linkinghub.elsevier.com/retrieve/pii/S0021999106001926

[57] N. Hoppe, S. Adami, N. A. Adams, Computer Methods in Applied Mechanics and Engineering 391 (2022) 1–61. doi:10.1016/j.cma.2021.114486.

[58] X. Y. Hu, B. C. Khoo, Journal of Computational Physics 198 (1) (2004) 35–64. doi:10.1016/j.jcp.2003.12.018.
URL http://linkinghub.elsevier.com/retrieve/pii/S0021999104000178

[59] R. R. Nourgaliev, T. G. Theofanous, Journal of Computational Physics 224 (2) (2007) 836–866. doi:10.1016/j.jcp.2006.10.031.
URL http://linkinghub.elsevier.com/retrieve/pii/S0021999106005511

[60] G. Russo, P. Smereka, Journal of Computational Physics 163 (2000) 51–67. doi:10.1006/jcph.2000.6553.
URL http://www.idealibrary.com

[61] G. S. Jiang, D. Peng, SIAM Journal on Scientific Computing 21 (6) (2000) 2126–2143. doi:10.1137/S106482759732455X.
URL http://www.siam.org/journals/ojsa.php

[62] E. F. Toro, Shock Waves 29 (8) (2019) 1065–1082. doi:10.1007/s00193-019-00912-4.
URL https://doi.org/10.1007/s00193-019-00912-4

[63] N. Fleischmann, S. Adami, N. A. Adams, Journal of Computational Physics 423 (2020) 109762. doi:10.1016/j.jcp.2020.109762.
URL https://doi.org/10.1016/j.jcp.2020.109762

[64] S. F. Davis, SIAM Journal on Scientific and Statistical Computing 9 (3) (1988) 445–473. doi:10.1137/0909030.
URL http://epubs.siam.org/doi/10.1137/0909030

[65] B. Einfeldt, SIAM Journal on Numerical Analysis 25 (2) (1988) 294–318. doi:10.1137/0725021.

[66] F. Acker, R. B. Borges, B. Costa, Journal of Computational Physics 313 (2016) 726–753. doi:10.1016/j.jcp.2016.01.038.
URL http://dx.doi.org/10.1016/j.jcp.2016.01.038

[67] N. R. Gande, A. A. Bhise, Numerical Algorithms (2020). doi:10.1007/s11075-020-01039-9.

[68] R. Borges, M. Carmona, B. Costa, W. S. Don, Journal of Computational Physics 227 (6) (2008) 3191–3211. doi:10.1016/j.jcp.2007.11.038.

[69] X. Y. Hu, Q. Wang, N. A. Adams, Journal of Computational Physics 229 (23) (2010) 8952–8965. doi:10.1016/j.jcp.2010.08.019.
URL http://linkinghub.elsevier.com/retrieve/pii/S0021999110004560

[70] X. Hu, N. Adams, Journal of Computational Physics 230 (19) (2011) 7240–7249. doi:10.1016/j.jcp.2011.05.023.
URL http://linkinghub.elsevier.com/retrieve/pii/S0021999111003342

[71] D. S. Balsara, C. W. Shu, Journal of Computational Physics 160 (2) (2000) 405–452. doi:10.1006/jcph.2000.6443.

[72] L. Fu, X. Y. Hu, N. A. Adams, Journal of Computational Physics 305 (2016) 333–359. doi:10.1016/j.jcp.2015.10.037.

[73] S. Hickel, C. P. Egerer, J. Larsson, Physics of Fluids 26 (10) (2014). doi:10.1063/1.4898641.
URL http://dx.doi.org/10.1063/1.4898641

[74] R. Menikoff, B. J. Plohr, Reviews of Modern Physics 61 (1) (1989) 75–130. doi:10.1103/RevModPhys.61.75.

[75] R. P. Fedkiw, T. Aslam, B. Merriman, S. Osher, Journal of Computational Physics 152 (2) (1999) 457–492. doi:10.1006/jcph.1999.6236.

[76] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Nature 585 (7825) (2020) 357–362. doi:10.1038/s41586-020-2649-2.
URL https://doi.org/10.1038/s41586-020-2649-2

[77] S. Van Der Walt, S. C. Colbert, G. Varoquaux, Computing in Science and Engineering 13 (2) (2011) 22–30. doi:10.1109/MCSE.2011.37.
URL http://ieeexplore.ieee.org/document/5725236/

[78] T. Hennigan, T. Cai, T. Norman, I. Babuschkin (2020).
URL http://github.com/deepmind/dm-haiku

[79] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener, T. Hennigan (2020).
URL http://github.com/deepmind/optax

[80] D. P. Kingma, J. L. Ba, in: 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings, International Conference on Learning Representations, ICLR, 2015. arXiv:1412.6980.

[81] G. A. Sod, Journal of Computational Physics 27 (1) (1978) 1–31. doi:10.1016/0021-9991(78)90023-2.

[82] P. D. Lax, Communications on Pure and Applied Mathematics 7 (1) (1954) 159–193. doi:10.1002/cpa.3160070112.

[83] U. Ghia, K. N. Ghia, C. T. Shin, Journal of Computational Physics 48 (3) (1982) 387–411. doi:10.1016/0021-9991(82)90058-4.

[84] L. Vinet, A. Zhedanov, A 'missing' family of classical orthogonal polynomials, Vol. 44, Cambridge University Press, 2011. doi:10.1088/1751-8113/44/8/085201.

[85] E. T. Spyropoulos, G. A. Blaisdell, AIAA Journal 34 (5) (1996) 990–998. doi:10.2514/3.13178.
URL https://arc.aiaa.org/doi/10.2514/3.13178

[86] K. Peery, S. Imlay, in: 24th Joint Propulsion Conference, American Institute of Aeronautics and Astronautics, Reston, Virigina, 1988. doi:10.2514/6.1988-2904.
URL https://arc.aiaa.org/doi/10.2514/6.1988-2904

[87] L. Rayleigh, Proceedings of the Royal Society of London 29 (196-199) (1879) 71–97. doi:10.1098/rspl.1879.0015.

[88] I. Taylor, G., Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character 146 (858) (1934) 501–523. doi:10.1098/rspa.1934.0169.

[89] H. Terashima, G. Tryggvason, Journal of Computational Physics 228 (11) (2009) 4012–4037. doi:10.1016/j.jcp.2009.02.023.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999109000898

[90] J. F. Haas, Journal of Fluid Mechanics 181 (1987) 41–76. doi:10.1017/S0022112087002003.

[91] M. E. Brachet, D. Meiron, S. Orszag, B. Nickel, R. Morf, U. Frisch, Journal of Statistical Physics 34 (5-6) (1984) 1049–1063. doi:10.1007/BF01009458.

[92] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
URL https://www.deeplearningbook.org/

[93] J. Duchi, E. Hazan, Y. Singer, in: COLT 2010 - The 23rd Conference on Learning Theory, Vol. 12, 2010, pp. 257–269.
URL https://dl.acm.org/doi/pdf/10.5555/1953048.2021068

[94] A. Harten, Journal of Computational Physics 115 (2) (1994) 319–338. doi:10.1006/jcph.1994.1199.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999184711995

[95] A. Harten, Communications on Pure and Applied Mathematics 48 (12) (1995) 1305–1342. doi:10.1002/cpa.3160481201.
URL http://doi.wiley.com/10.1002/cpa.3160481201http://onlinelibrary.wiley.com/doi/10.1002/cpa.3160481201/abstract

[96] J. Romero, J. Crabill, J. E. Watkins, F. D. Witherden, A. Jameson, Computer Physics Communications 250 (2020). doi:10.1016/j.cpc.2020.107169.

[97] D. Häfner, F. Vicentini, Journal of Open Source Software 6 (65) (2021) 3419. doi:10.21105/joss.03419.
URL https://joss.theoj.org/papers/10.21105/joss.03419

[98] G. Henry, P. T. P. Tang, A. Heinecke, Proceedings - Symposium on Computer Arithmetic 2019-June (2019) 69–76. arXiv:1904.06376, doi:10.1109/ARITH.2019.00019.

[99] S. Williams, A. Waterman, D. Patterson, Communications of the ACM 52 (4) (2009) 65–76. doi:10.1145/1498765.1498785.
URL https://dl.acm.org/doi/10.1145/1498765.1498785

[100] C. Yang (2020). arXiv:2009.02449.
URL http://arxiv.org/abs/2009.02449

[101] A. Khajeh-Saeed, J. Blair Perot, Journal of Computational Physics 235 (2013) 241–257. doi:10.1016/j.jcp.2012.10.050.
URL https://linkinghub.elsevier.com/retrieve/pii/S0021999112006547

Declaration of interests

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

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: