Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing

Hendrik Ranocha[1], Michael Schlottke-Lakemper[2, 3], Andrew R. Winters[4], Erik Faulhaber[2], Jesse Chan[5], and Gregor J. Gassner[2, 3]

[1]Applied Mathematics, University of Münster, Germany
[2]Department of Mathematics and Computer Science, University of Cologne, Germany
[3]Center for Data and Simulation Science, University of Cologne, Germany
[4]Department of Mathematics; Applied Mathematics, Linköping University, Sweden
[5]Department of Computational and Applied Mathematics, Rice University, Houston, Texas, US

ABSTRACT

We present Trixi.jl, a Julia package for adaptive high-order numerical simulations of hyperbolic partial differential equations. Utilizing Julia's strengths, Trixi.jl is extensible, easy to use, and fast. We describe the main design choices that enable these features and compare Trixi.jl with a mature open source Fortran code that uses the same numerical methods. We conclude with an assessment of Julia for simulation-focused scientific computing, an area that is still dominated by traditional high-performance computing languages such as C, C++, and Fortran.

Keywords

Julia, Scientific Computing, Numerical Simulations, Conservation Laws, Discontinuous Galerkin Methods, Adaptive Mesh Refinement, Compressible Euler Equations, Ideal Magnetohydrodynamics, Entropy Stability, Shock Capturing

1. Introduction

We are broadly interested in simulation-focused scientific computing, in particular numerical approximations for hyperbolic partial differential equations (PDEs), computational fluid dynamics (CFD), and related problems. The focus of our research ranges from specific applications in CFD to general multi-physics coupling strategies to the development and analysis of the high-order numerical methods our simulations are built upon. In addition, we are academics and involved in teaching students in these areas of science. Thus, we would like to have a code that is

(1) extensible for research and development,

(2) easy to understand and use for students and collaborators,

(3) fast enough for applied 3D problems.

Additionally, we are greedy[1] and wish to include all these features within a single code. Roughly one year of collaborative work has resulted in the current version of Trixi.jl[2], providing adaptive high-order numerical simulations of hyperbolic PDEs in Julia [5]. Starting as an experiment, we have been able to reach more and more of our goals with Trixi.jl.

In this article, we present an overview of the main features and design decisions of Trixi.jl in Section 2, laying the ground for an extensible and easy-to-use framework of high-order methods for hyperbolic PDEs. Next, we compare the serial performance with a mature high-performance computing (HPC) Fortran code in Section 3, demonstrating that Julia is not generically slower than traditional HPC languages (and can even be faster in this particular case). Thereafter, we present an assessment of Julia for simulation-focused scientific computing based on our experience with Trixi.jl in Section 4. Finally, we summarize our findings and conclusions in Section 5.

2. Capabilities and design of Trixi.jl

Trixi.jl is designed as a simulation framework and library of high-order methods for conservation laws of the form

$$\partial_t u(t,x) + \sum_{j=1}^{d} \partial_{x_j} f^j(u) = s(t,x,u), \quad t \in (0,T), x \in \Omega, \quad (1)$$

in $d \in \{1,2,3\}$ space dimensions. Here, the independent variables are time t and space coordinates $x \in \Omega \subset \mathbb{R}^d$. The conserved quantities are denoted as u, e.g., mass, momentum, and energy for the compressible Euler equations of an ideal gas. The physical system is specified by the fluxes f^j and the source term s. In addition, suitable initial and boundary conditions (ICs, BCs) are required. Trixi.jl also handles non-conservative PDE terms as in the shallow water equations or magnetohydrodynamics equations with divergence cleaning, where source terms can depend on derivatives of u.

2.1 Main features of Trixi.jl

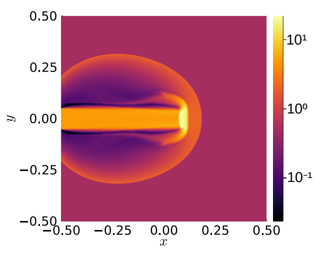
As of version v0.3.55 (August 2021), Trixi.jl concentrates mainly on discontinuous Galerkin (DG) methods [22, 27]. In particular, it has a focus on entropy-conservative and -dissipative methods [51, 29, 15, 39, 10]. Currently, Trixi.jl offers the following features:

[1]See `https://julialang.org/blog/2012/02/why-we-created-julia` (accessed 2021-08-11)

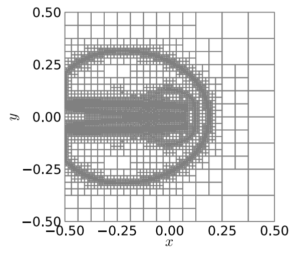
[2]`https://github.com/trixi-framework/Trixi.jl`

- 1D, 2D, and 3D simulations on line/quad/hex/simplex meshes
  - Cartesian and curvilinear meshes
  - Conforming and non-conforming meshes
  - Structured and unstructured meshes
  - Hierarchical quadtree/octree grid with adaptive refinement
  - Forests of quadtrees/octrees with `p4est` [7] via P4est.jl[3]
- High-order matrix-free discontinuous Galerkin methods
  - Kinetic energy preserving and entropy-stable methods
  - Entropy-stable sub-cell shock capturing
  - Sub-cell positivity-preserving limiting
- Multiple governing equations
  - Compressible Euler equations (optionally with self-gravity)
  - Magnetohydrodynamics (MHD) equations
  - Multicomponent compressible Euler and MHD equations
  - Acoustic perturbation equations
  - Hyperbolic diffusion for elliptic problems
  - Lattice-Boltzmann equations (D2Q9 and D3Q27 schemes)
  - Several scalar conservation laws (e.g., advection, Burgers)
- Integration with the Julia package ecosystem and external tools
  - Time integration methods from OrdinaryDiffEq.jl
  - Automatic differentiation with ForwardDiff.jl
  - In-situ visualization with Plots.jl
  - Postprocessing with ParaView/Visit via Trixi2Vtk.jl[4]

Currently, Trixi.jl provides shared-memory parallelization via multithreading. Initial parallelization with MPI is available for some mesh types, but full support for distributed-memory parallelism is subject to ongoing work.
Some of the main features of Trixi.jl are demonstrated in the following figures. The detailed physical setups and numerical parameters as well as all code necessary to reproduce these figures are available in the reproducibility repository for this article [47].
Figure 1 demonstrates sub-cell entropy-dissipative shock-capturing methods with sub-cell positivity-preserving limiters and adaptive mesh refinement applied to an astrophysical supersonic jet with Mach number 2000 [32].



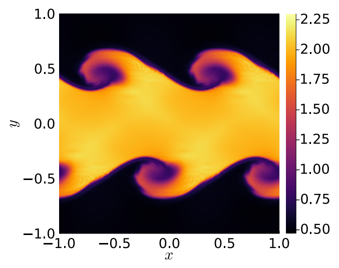
(a) Density at time $t = 10^{-3}$.



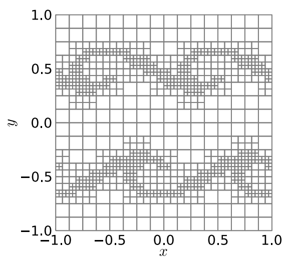
(b) Mesh at time $t = 10^{-3}$.

Fig. 1: Numerical solutions of a supersonic jet with Mach number 2000 using sub-cell entropy-dissipative shock-capturing methods with sub-cell positivity-preserving limiters and adaptive mesh refinement for the compressible Euler equations.

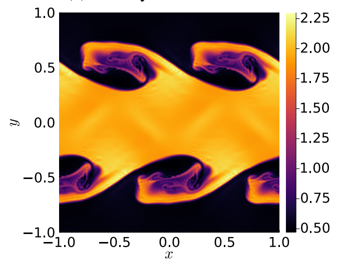
Figure 2 demonstrates nonlinear stability obtained with entropy-stable methods and adaptive mesh refinement applied to a classical Kelvin-Helmholtz flow instability problem.
Figure 3 shows the approximation of acoustic perturbation equations [14] wave scattering on a curvilinear and unstructured domain. The 2D quadrilateral mesh used in the simulation (Figure 3d) was generated with HOHQMesh.jl[5].



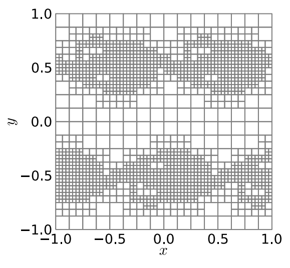
(a) Density at time $t = 2$.



(b) Mesh at time $t = 2$.

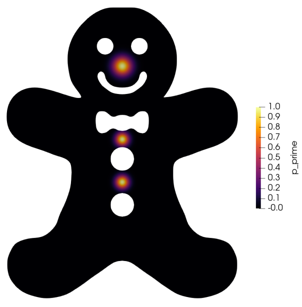


(c) Density at time $t = 3$.

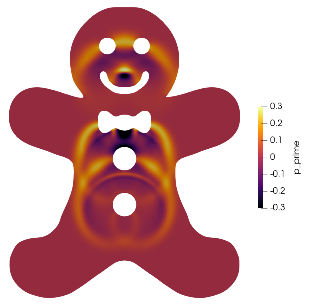


(d) Mesh at time $t = 3$.

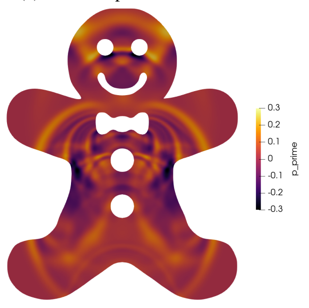
Fig. 2: Numerical solutions of a Kelvin-Helmholtz instability using entropy-stable methods and adaptive mesh refinement for the compressible Euler equations.



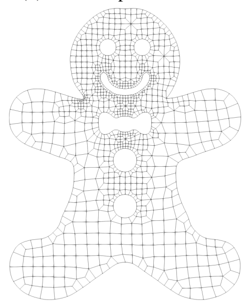
(a) Acoustic pressure at time $t = 0$.



(b) Acoustic pressure at time $t = 8$.



(c) Acoustic pressure at time $t = 16$.



(d) Unstructured quad mesh.

Fig. 3: Numerical solutions of pressure wave scattering for the acoustic perturbation equations at three points in time as well as the unstructured, curvilinear quadrilateral mesh.

2.2 High-level overview of the code structure

Trixi.jl is built from the method of lines. Thus, a discretization of (1) is obtained in two steps. First, a spatial semidiscretization is created. Next, the resulting ordinary differential equation (ODE) is solved using a time integration method. Currently, Trixi.jl focuses on the spatial semidiscretization and uses mostly Runge-Kutta methods implemented in OrdinaryDiffEq.jl, which is part of DifferentialEquations.jl [37].

[3]`https://github.com/trixi-framework/P4est.jl`
[4]`https://github.com/trixi-framework/Trixi2Vtk.jl`
[5]`https://github.com/trixi-framework/HOHQMesh.jl`

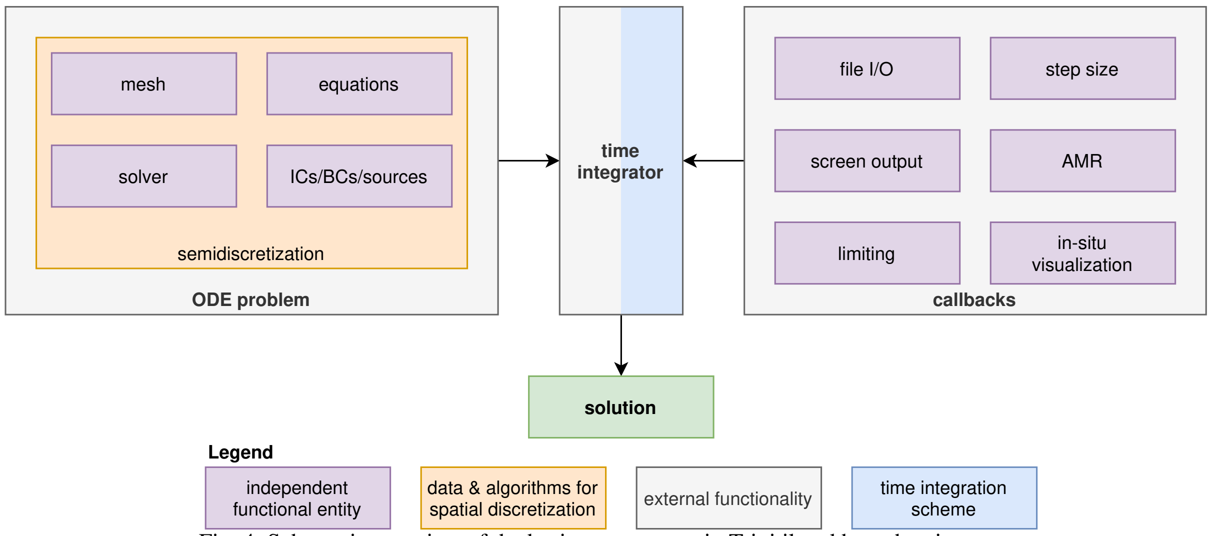


Fig. 4: Schematic overview of the basic components in Trixi.jl and how they interact.

Figure 4 presents an overview of the basic components of Trixi.jl. The most important structure is the semidiscretization, which bundles all information about the spatial approximation. The mathematical-physical model is determined by the `equations`, the `initial_condition`, `boundary_conditions`, and possible `source_terms`. The `solver` describes purely numerical parameters determining the specific discretization method such as discontinuous Galerkin or finite difference methods, kinetic energy preserving or shock-capturing approaches. Finally, the `mesh` has a necessarily hybrid role including information about the spatial domain and its discretization.
Once information about the time span $[0, T]$ is provided, a semidiscretization in Trixi.jl can be converted into an ODE problem, which can be solved by methods from OrdinaryDiffEq.jl. The flexible callback infrastructure of this ODE library allows us to provide extended functionality for Trixi.jl without modifying any main loop. In particular, various tasks such as input/output operations, adaptive mesh refinement (AMR), and positivity-preserving limiting are implemented using callbacks. In case full control over the time step loop is required, e.g., to experiment with new features that are not easily realized with the existing API, Trixi.jl also implements its own time integration schemes that mimic the interface of OrdinaryDiffEq.jl and that can be used as drop-in replacements.

2.3 Review of the main design choices

The design of Trixi.jl was guided by extensibility, ease of use, and efficiency (in this particular order). Thus, it is likely possible to make some parts even faster at the cost of simplicity and readability. Nevertheless, Trixi.jl is already quite fast, even compared to mature high-performance open source software, as described in detail in Section 3. In this section, we focus mostly on the first two goals.

2.3.1 Trixi.jl is a library. The most important design decision was to create Trixi.jl as a library. In the context of simulation-focused scientific computing, there are many (open or closed source) codes that are designed as monolithic applications that can simulate a specific setup. These codes can often be configured to a limited extent by specifying compile time options or editing parameter files. In contrast, users of Trixi.jl do not need to compile anything or learn parameter file syntax. Instead, they can just load the package and write their simulation setups in Julia. Scripts that describe a simulation setup for Trixi.jl are referred to as “elixirs”. As of version v0.3.55 (August 2021), Trixi.jl comes with approximately 200 example elixirs. A basic elixir looks as follows:

```julia
# Load all required libraries
using Trixi, OrdinaryDiffEq, Plots

# Set up a 2D linear advection problem with
# advection velocity `a`
a = (1.0, 1.0)
equations = LinearScalarAdvectionEquation2D(a)

# Choose an initial condition coming with Trixi.jl
ic = initial_condition_convergence_test

# Set up a discontinuous Galerkin spectral element
# method with given polynomial degree and surface
# flux implemented in Trixi.jl
solver = DGSEM(polydeg=3,
               surface_flux=flux_lax_friedrichs)

# Create a uniformly refined mesh with periodic
# boundary conditions in a square domain
coordinates_min = (-1.0, -1.0) # lower left
coordinates_max = ( 1.0,  1.0) # upper right
mesh = TreeMesh(coordinates_min, coordinates_max,
                initial_refinement_level=4,
                n_cells_max=10^5, periodicity=true)

# Create semidiscretization with all spatial
# discretization-related components
semi = SemidiscretizationHyperbolic(
    mesh, equations, ic, solver)

# Create an ODE problem from the semidiscretization
# with time span from 0.0 to 1.0
ode = semidiscretize(semi, (0.0, 1.0))

# Evolve the ODE problem in time using `solve` from
# OrdinaryDiffEq with adaptive time stepping
sol = solve(ode, RDPK3SpFSAL49(), abstol=1.0e-6,
    reltol=1.0e-6, save_everystep=false)

# Plot the numerical solution at the final time
plot(sol)
```

2.3.2 Functions are pure Julia functions. Since functions are first-class citizens in Julia, they can be passed around and used efficiently. Thus, everything that acts like a function can be used in Trixi.jl, e.g., to define initial/boundary conditions. While the flexibility to change initial/boundary conditions is quite common, two-point numerical fluxes are also just functions with a specified signature. Thus, users can implement numerical fluxes in their own code and they will work and be as efficient as if they were implemented directly in Trixi.jl.

2.3.3 Ingredients use common interfaces and can be exchanged. Abstractions such as (variants of) the `solver`, the `mesh`, and the `equations` use a common interface. Utilizing multiple dispatch in Julia, internal implementations are specialized accordingly. For example, changing the volume terms of the DG discretization from the standard weak form to entropy-stable and/or shock-capturing methods can be achieved by passing one additional parameter to the `DGSEM` constructor in the example above. Moreover, there is no hidden global state. This means that multiple instances of similar structures can be instantiated simultaneously. In particular, multiple semidiscretizations (in possibly different spatial dimensions) can be created and used in the same code.

2.3.4 New physics can be specified with minimal effort. Instead of providing only the very basic discretization ingredients, as with other open source libraries for PDEs, Trixi.jl also includes some widely used physical systems as well as analysis routines, such as computation of the integrated kinetic energy for the compressible Euler equations, to make it easy to use out of the box. Nevertheless, users are not restricted to the physics models bundled in Trixi.jl. To set up a new type of `equations`, it is only necessary to create an appropriate `struct` containing all parameters and to implement pointwise operations such as the calculation of the fluxes f^j in (1) or two-point numerical fluxes. Due to the Julia language design with just-ahead-of-time compilation, these fluxes can be inlined into the library functions of Trixi.jl. Thus, the physics is completely separated from the `solver` but remains computationally efficient. In particular, this allows a user to reuse the same numerical fluxes for discontinuous Galerkin methods, finite difference methods, and variants of finite volume methods. Due to favoring simplicity over excessively generic code, it remains comparatively straightforward to modify existing Trixi.jl implementations if a new feature requires modifications to methods.

2.3.5 There is no spooky action at a distance. In monolithic code bases, it is often necessary to implement a new feature for all combinations of possible (compile time) options. This effort can make it difficult to experiment with new ideas. In contrast, Julia's dynamic nature, multiple dispatch, and just-ahead-of-time compilation allow us to implement only those features that are strictly necessary. For example, if a user wants to simulate a new physics model only on Cartesian grids with smooth solutions, there is no need to implement anything for curvilinear coordinates or shock-capturing approaches. In addition, it allows a user to extend Trixi.jl step-by-step with new capabilities. For example, there are ongoing efforts to incorporate summation-by-parts finite difference methods via SummationByPartsOperators.jl [43] and discontinuous Galerkin methods on simplex elements via StartUpDG.jl[6] in Trixi.jl. This is feasible because the modular design of Trixi.jl gives users the flexibility to pick a subset of the available meshes/methods, selected via multiple dispatch, to test their newly implemented features.

[6] `https://github.com/jlchan/StartUpDG.jl`

3. Performance comparison with Fortran

FLUXO[7] is an open source Fortran code implementing discontinuous Galerkin methods on unstructured hexahedral meshes in 3D for advection-diffusion equations. It provides the same kind of modern, flux differencing DG methods to achieve entropy conservation/dissipation or kinetic energy preservation that are used in Trixi.jl. At the same time, both codes have capabilities the other does not, e.g., support for parabolic equations and multi-node parallelism in FLUXO, or multiple mesh types and more physics setups in Trixi.jl. Nevertheless, both Trixi.jl and FLUXO share a common set of features, i.e., high-order DG methods on 3D curvilinear meshes, which allows a reasonable comparison of their performance.

3.1 Description of the setup

Here, we compare the serial performance of Trixi.jl and FLUXO when solving a hyperbolic PDE in three space dimensions on curvilinear hexahedral meshes.
For the problem setup we consider a periodic box of the domain $[-1, 1]^3$ with four elements in each spatial direction. This results in $64(\texttt{polydeg} + 1)^3$ degrees of freedom (DG nodes) per equation when polynomials of degree `polydeg` are used. To make the mesh curvilinear, the interior of the box is heavily warped by a mapping adapted from [9]. Specifically, we map Cartesian reference coordinates $(\xi, \eta, \zeta) \in [-1, 1]^3$ to physical coordinates (x, y, z) via the transformation

$$\begin{aligned} y &= \eta + 0.15\big(\cos(1.5\pi\xi)\cos(0.5\pi\eta)\cos(0.5\pi\zeta)\big), \\ x &= \xi + 0.15\big(\cos(0.5\pi\xi)\cos(2\pi y)\cos(0.5\pi\zeta)\big), \\ z &= \zeta + 0.15\big(\cos(0.5\pi x)\cos(\pi y)\cos(0.5\pi\zeta)\big). \end{aligned} \tag{2}$$

To integrate up to the final time $T = 1.0$, both codes use the five-stage, fourth-order low-storage explicit Runge-Kutta method of Carpenter and Kennedy [26]. A stable explicit time step is adaptively computed according to the local maximum wave speed, the relative grid size, and an adjustable Courant-Friedrichs-Lewy (CFL) coefficient $\text{CFL} \in (0, 1]$ [19]. For the performance computations presented herein we fix this coefficient to be $\text{CFL} = 0.5$.
We compare the performance of FLUXO and Trixi.jl for two smooth nonlinear problems, a manufactured solution for the 3D compressible Euler equations and Alfvén wave propagation for the 3D ideal magnetohydrodynamics (MHD) equations [18, 1]. These initial conditions, available in FLUXO and Trixi.jl, are typically used to demonstrate the high-order accuracy and convergence properties of the frameworks. The curved mesh and the initial density for the compressible Euler problem are visualized in Figure 5.
For these comparisons we examine the performance of FLUXO and Trixi.jl for the weak form as well as flux differencing implementations of the DG solver. For the weak form simulations we compute the coupling between elements with a Harten-Lax-van Leer (HLL) numerical surface flux function [52, 31]. For the flux differencing solver, which introduces an additional numerical volume flux function, we use an entropy-conservative flux function in both the volume and at the surface. For the compressible Euler equations this flux is also kinetic energy preserving [41, 40] while for the ideal MHD equations it is kinetic and magnetic energy preserving [23]. We note that both ideal MHD implementations require additional computational effort due to non-conservative terms necessary for the numerical approximation to remain entropy consistent [6]. Additionally, the weak form DG solver type is recovered if one uses a

[7] `https://gitlab.com/project-fluxo/fluxo`

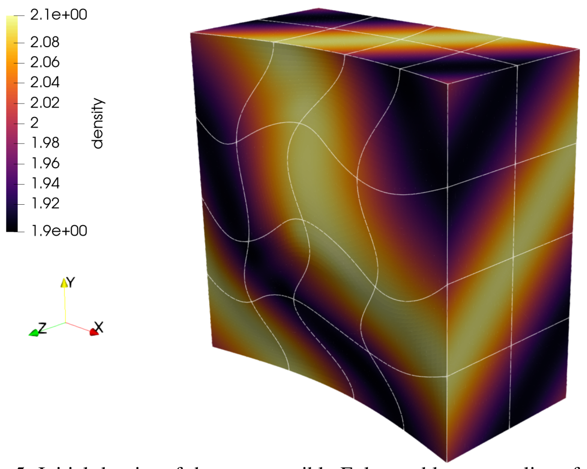


Fig. 5: Initial density of the compressible Euler problem on a slice of the periodic, curved mesh used for the performance benchmarks.

central flux in the volume of the flux differencing DG solver, albeit with additional computational cost [20].

All simulations for the performance comparison were run on the HPC system Tetralith provided by the Swedish National Infrastructure for Computing (SNIC). Each node of Tetralith has two sockets, each with an Intel(R) Xeon(R) Gold 6130 @ 2.10GHz with 96GiB of memory. FLUXO was compiled using the Intel compiler suite v18.0.3 and Trixi.jl used Julia v1.6.1. The serial performance results were obtained on a single core. We ran each configuration of a particular physical setup, polynomial degree, and solver type for each code five times, taking the smallest value from each for the comparison results. We ran the same PID tests in FLUXO compiled with GCC v6.4.0 but found that the version compiled using Intel was between 5 % to 23 % faster. Thus, only results from the Intel compiler are included.

To assess the performance of the two codes, we vary the polynomial degree (`polydeg`) from 3 to 15. This corresponds to considering DG approximations of increasing spatial accuracy, from fourth up to sixteenth order. The metric we use to analyze and compare FLUXO and Trixi.jl simulations is a performance index (PID) that measures the time required to advance a single degree of freedom (DOF) from one stage of the explicit Runge-Kutta time integration scheme to the next. It is computed as

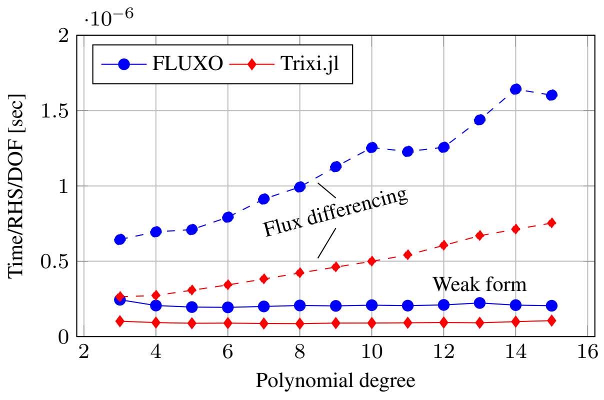
$$\mathrm{PID} = \frac{\text{wall-clock time}}{\#\text{time steps} \cdot 5 \cdot \#\text{elements} \cdot (\texttt{polydeg} + 1)^3}, \qquad (3)$$

where 5 is the number of Runge-Kutta stages per time step for the selected time-stepping method. In other words, the PID measures the run time required for the right hand side (RHS) evaluation of each DOF.

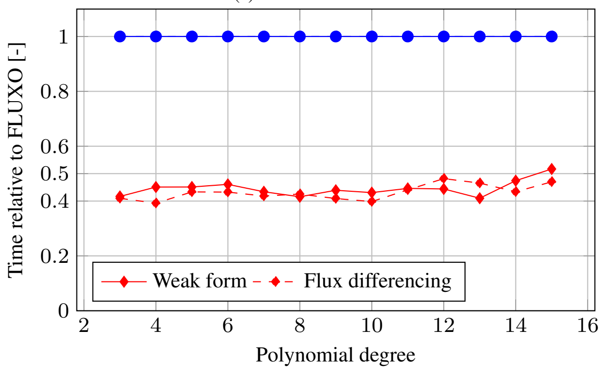
The code and detailed information necessary to reproduce these numerical experiments are available in the accompanying repository [47].

3.2 Results of the performance comparison

The results of these performance measurements are visualized for the 3D compressible Euler equations in Figure 6 and for the 3D ideal MHD equations in Figure 7. We present the absolute PID timings for both codes in the top portion of these two figures and a relative comparison using FLUXO as reference in the bottom portion; smaller values thus are always better.



(a) Absolute run times.

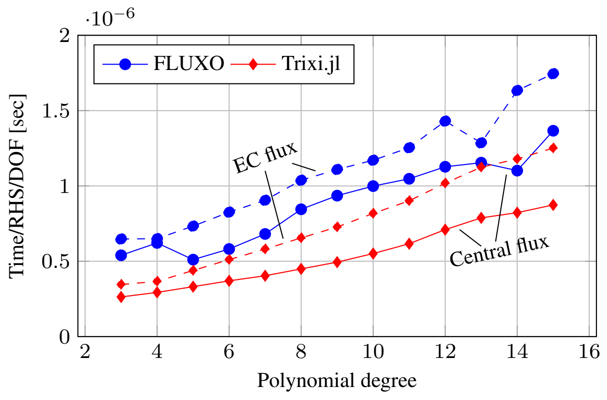


(b) Run time relative to FLUXO.

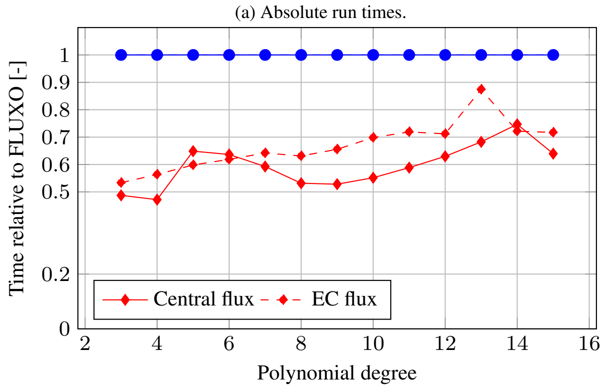
Fig. 6: Run time per right-hand side evaluation and degree of freedom for different DG discretizations of the 3D compressible Euler equations in FLUXO (Fortran) and Trixi.jl (Julia).

From these results, it is clear that Trixi.jl is more than 2x faster for the compressible Euler equations and more than 1.5x faster for the ideal MHD equations than the Fortran code FLUXO. It is also possible to see the additional computational effort for the entropy-conservative flux differencing DG methods for both physical systems. This is because the numerical fluxes are more expensive in terms of computational cost, and require special care to optimize their performance [42, 46]. We reiterate that both codes implement the same numerical nodal DG methods on curvilinear meshes used in these tests. This demonstrates the suitability of Julia for this kind of simulation-focused scientific computing.

While our Julia code is faster than the mature HPC Fortran code FLUXO for this non-trivial example, we do not claim that Julia is generally faster than Fortran, C, or C++. Instead, we would like to emphasize that well-written Julia code can be *at least as fast* as code written in these traditional scientific computing languages, as demonstrated also by several microbenchmarks [5]. Further, we expect to be able to achieve similar performance with either language by spending enough time and effort to optimize the respective codes. Trixi.jl owes its performance optimizations in part to the code introspection and profiling tools available in Julia. Similar tools used to optimize the performance in other languages are often not as easy to use as their Julia counterparts. Therefore, we developed some additional performance improvements in Trixi.jl and ported them to FLUXO later [46]. Moreover, Trixi.jl uses slightly different implementations of computationally intensive kernels and stores marginally less information in memory, recomputing some



(a) Absolute run times.



(b) Run time relative to FLUXO.

Fig. 7: Run time per right-hand side evaluation and degree of freedom for the flux differencing DG discretization of the 3D ideal MHD equations. Two configurations are compared using either the central volume flux (algebraically equivalent to the weak form DG solver) and the entropy conservative (EC) volume flux.

terms instead. Additionally, the manufactured solution of the compressible Euler equations uses source terms containing sine and cosine terms, which are computed together by `sincos` in Trixi.jl but require individual calls to `sin` and `cos` in FLUXO.
Finally, the different mesh types available in Trixi.jl allow further optimizations. For example, if the computational domain is essentially a cube (or square, line) without need for curvilinear coordinates, the Cartesian `TreeMesh` can be used. Depending on the particular choice of discretization methods, the Cartesian `TreeMesh` can be 10 % to 25 % more efficient than the curvilinear meshes in Trixi.jl.

4. Assessment of Julia for simulation-focused scientific computing

Julia was designed from scratch to be suitable for numerical computing [5]. Its usefulness has been demonstrated, e.g., for GPU programming [4, 34], optimization [13], data science including big data[8], machine learning [25], and scientific machine learning (SciML) [35]. There are also mature libraries for ODE problems [37], small scale spectral approximations [33], and classical finite element methods [2]. In addition, there are some packages dedicated to solving specific time-dependent PDEs such as [38, 11]. However, there does not seem to be a general framework for high-order methods for hyperbolic PDEs in Julia. Thus, Trixi.jl is an ideal candidate for a case study of Julia for simulation-focused scientific computing, an area where codes written in classical programming languages such as C and Fortran are still dominant [28, 36].

4.1 What works well

4.1.1 Julia is fast. As demonstrated in Section 3, Julia is not generically slower than traditional high-performance programming languages like C, C++, and Fortran. A minor exception in the context of scientific computing relying on floating point operations is that Julia does not perform automatic fused multiply-add (FMA) contraction, which can be remedied by explicitly using the `muladd`[9] function or the more convenient `@muladd`[10] macro (for more details see the appendix).

4.1.2 Julia encourages good software development practices. Many scientists implementing numerical methods have no formal training in software development. Julia and its ecosystem support these researchers by making it easy to set up unit and regression tests, since a testing framework is included in the standard library. This facilitates test-driven development and continuous integration (CI), which makes code restructuring and optimization straightforward.
Julia itself and most packages are developed on GitHub. The Julia community provides several tools to run corresponding CI setups, prepare and publish documentation[11], and other related tasks. This setup also encourages other good software development practices that we use in Trixi.jl, such as mandatory code reviews.
In addition, the Julia package manager `Pkg` fosters a unified form of semantic versioning across the Julia ecosystem, which lays the basis for the following three observations.

4.1.3 It is easy to set up reproducible numerical experiments. The package manager `Pkg` makes it easy to reproduce the exact runtime environment, including binary dependencies, used to generate numerical results. We have used this feature for all of our papers based on Trixi.jl [49, 45], including the present manuscript [47]. This facilitates code sharing and reproducible research in computational science, which is arguably important but not yet mainstream [3, 12, 30].

4.1.4 External libraries can be integrated with relative ease. To incorporate some of the adaptive mesh capabilities into Trixi.jl we have created a Julia wrapper[12] for the C library `p4est` [7]. In our experience, it is relatively easy to do so in Julia, and the package manager in combination with BinaryBuilder.jl makes it convenient to distribute the necessary binaries. In particular, users do not have to compile libraries on their own system and the binaries are available on all major platforms including Linux, macOS, and Windows. Similarly, there is no need to compile HDF5 on a user system due to the availability of HDF5.jl[13]. There is still work to be done on how to improve the experience in combination with MPI on HPC clusters, but some promising approaches exist (see Section *4.2.2*).

[8]Celeste project: `https://github.com/jeff-regier/Celeste.jl`, see `https://juliacomputing.com/case-studies/celeste` (accessed 2021-08-11)

[9]`https://docs.julialang.org/en/v1/base/math/#Base.muladd`
[10]`https://github.com/SciML/MuladdMacro.jl`
[11]`https://github.com/JuliaDocs/Documenter.jl`
[12]`https://github.com/trixi-framework/P4est.jl`
[13]`https://github.com/JuliaIO/HDF5.jl`

It is also possible to wrap Fortran libraries/tools in Julia. For example, we have created the wrappers KROME.jl[14] for KROME, a package to embed chemistry in astrophysical simulations [21], and HOHQMesh.jl[15] for HOHQMesh[16], a high order hex-quad mesh generator written in Fortran.

4.1.5 Packages can be used together at low (or no) cost. Due to Julia's package manager, the cost of using external dependencies is low enough to not become an issue. Traditionally, many scientific simulation codes tend to reduce the number of external dependencies as much as possible due to the complexity of handling different versions and making everything work together. Such an approach often leads to significant code duplication. For example, many CFD codes implement their own time integration methods, arguing that the spatial part is much more complex. While this is often true, it is still more efficient in terms of developer time to reuse existing implementations. This allows experts on time integration methods to develop specialized algorithms and implement them in open source software, while practitioners or researchers focusing on spatial semidiscretizations can benefit with near-zero effort. In our case, optimized time integration methods were developed in [44] and implemented in OrdinaryDiffEq.jl. These methods can be used with Trixi.jl by changing a single line of code. In contrast, researchers working without external dependencies would need extra time to digest a new method's details before even beginning an implementation. This additional overhead makes it less likely that researchers would use these novel schemes in their own codes and benefit from recent algorithmic developments.
An additional example is given by automatic differentiation. Due to multiple dispatch, it is possible to conveniently use forward mode automatic differentiation [48] to compute Jacobians of semidiscretizations of nonlinear conservation laws with Trixi.jl or to differentiate through a complete simulation, including time integration methods from OrdinaryDiffEq.jl.

4.1.6 Julia solves the two-language problem. Many libraries for simulation-focused scientific computing are written in languages such as C, C++, or Fortran to get decent performance. On top of these low-level details, high-level wrappers are often provided to make it easier for users to apply the algorithms for their problems. In contrast, Trixi.jl is written completely in Julia.
Such a difference between the programming languages used in the front end and back end leads to a well-known barrier between a user and the developers of a library. Julia was designed from the beginning to make it possible to write both simple high-level code and highly efficient compute kernels, thereby solving the two-language problem — at least to a significant extent. Highly efficient low-level code will still look different from simple high-level code, but the barrier between them is much smaller. In particular, this makes it easier for users to transition to developers and contribute to a package. This also contributes to the following observation:

4.1.7 The code base is simple enough to be useful for new users. Due to the package manager, Trixi.jl and all related ODE and visualization tools can be installed with a single command, reducing the overhead of exploring the code. In addition, Julia's expressiveness and the ability to work with a restricted subset of all possible features have enabled already more than 18 students to use Trixi.jl for their coursework or theses. In our experience, the effort to get started with traditional monolithic code bases is often so large that due to time restrictions students either choose to build their own implementations specialized on their tasks or only work on projects with a limited scope.
Another anecdotal example for the ease of use is the preprint [50]. While the authors utilized Trixi.jl for some numerical experiments, they did so independently of the development team (who learned that the authors were using Trixi.jl only *after* reading the preprint).

4.1.8 Existing features can be extended and combined easily. Due to Julia's high-level programming approach, dynamic typing, and multiple dispatch, it is easy to combine existing functionality efficiently. For example, the single-physics solvers for hyperbolic PDEs of Trixi.jl were extended to a multi-physics setup for the compressible Euler equations with self-gravity with roughly 350 lines of code [49]. In addition, Trixi.jl can be extended from the outside without modifying the main source code, which makes it easy to set up new simulation approaches and analyze existing ones, e.g., by fluctuation simulations [45].

4.1.9 Julia is free. Julia itself is released under the MIT license and ships some GPL-licensed third-party software (that can optionally be disabled for commercial purposes). Many packages in the Julia ecosystem follow this approach and are freely available under the permissive MIT license, including Trixi.jl. This allows programmers to use Julia without needing to pay for commercial software. In particular, students can work with the software on their private computers without restrictions.

4.2 What is still difficult or unknown

4.2.1 Compilation times can be annoying. Since Julia uses a serial just-ahead-of-time compiler that currently does not cache code between different Julia sessions, the initial compilation time can be annoying. For example, the time to finish a first Trixi.jl simulation in a fresh Julia session and plot the results on a notebook can take between 30 seconds and a minute. Having compiled the code, the second simulation and plot take less than 0.05 seconds. Thus, switching to Julia requires adapting the workflow compared to other languages such as keeping a REPL session active for a longer time and using packages such as Revise.jl[17]. This is particularly problematic in HPC environments. There are tools such as PackageCompiler.jl[18] that can help with these tasks but a good workflow usable for development and deployment still has to be found.

4.2.2 Simulations with distributed-memory parallelism. While Julia offers its own approach to distributed computing via the `Distributed` package in the standard library, its performance and scalability are currently limited [8]. As an alternative, the package MPI.jl utilizes the Message Passing Interface (MPI) for data exchange [8]. It provides an API similar to MPI's C interface and allows one to either wrap MPI binaries installed through Julia's package manager or to make use of an existing MPI installation. Most Julia projects focusing on distributed-memory simulations seem to be using MPI.jl, such as ClimateMachine.jl[19] or ImplicitGlobalGrid.jl[20]. In general, the performance of MPI programs written in Julia is comparable to C programs [24], and parallel simulations with Julia have been shown to scale to more than 5,000 GPUs with MPI.jl [34].

[14]`https://github.com/trixi-framework/KROME.jl`
[15]`https://github.com/trixi-framework/HOHQMesh.jl`
[16]`https://github.com/trixi-framework/HOHQMesh`
[17]`https://github.com/timholy/Revise.jl`
[18]`https://github.com/JuliaLang/PackageCompiler.jl`
[19]`https://github.com/CliMA/ClimateMachine.jl`
[20]`https://github.com/eth-cscs/ImplicitGlobalGrid.jl`

However, some practical challenges associated with running MPI-based parallel simulations with Julia remain. For starters, the issue of compilation times, described in the previous section, is exacerbated as MPI runs are by default not interactive and compilation results cannot be cached between MPI sessions. As such, this requires code to recompile for each subsequent execution of an MPI-parallel Julia program, which is particularly annoying during development. Partial relief can be obtained by using `tmpi`[21] that allows the user to simultaneously interact with multiple interactive Julia REPL sessions via the terminal multiplexer `tmux`. However, this approach does not scale beyond a few MPI ranks and is only available on Unix-like operating systems. The use of PackageCompiler.jl to reduce compilation times as much as possible thus becomes mandatory. Another issue encountered when running Julia with many parallel processes is that Julia operates with many small files during startup and precompilation, putting heavy pressure on the parallel file system. This can be partially alleviated by doing precompilation in serial. There are also tools like `Spindle`[22] that help solve similar problems that occur when running Python programs in parallel [16]. Finally, the ease of using external libraries as described in Section *4.1.4* is somewhat lost when working with dependencies that themselves are parallelized with MPI. In this case it is necessary to manually specify the paths to locally compiled libraries that have been built against the respective system MPI installation, since there is generally no binary compatibility between different MPI implementations. There are ongoing efforts in the Julia HPC community to work around this limitation by using the WI4MPI[23] wrapper package or the Spack package manager [17, 8]. Even though the aforementioned difficulties do not prevent massively parallel simulations with Julia, they raise the entry barrier for new users and developers. The overarching issue, however, is that at the time of writing, there is little precedence in terms of openly available tutorials, citable publications, or highly scalable example applications that can be used to learn about best practices for MPI-based parallelism in Julia. Thus, while in general it is justified to be cautiously optimistic that most issues can be overcome, it is too early to make a final assessment of the suitability of Julia for highly parallel simulations.

5. Summary and conclusions

We have presented Trixi.jl, a Julia package for adaptive high-order numerical simulations of hyperbolic PDEs. As researchers in numerical analysis and scientific computing, our goals were to create a framework that is extensible, easy to use, and fast (in this particular order). Making use of Julia's strengths, we have been successful based on recent publications making use of Trixi.jl, the number of students and researchers working with Trixi.jl, and serial performance comparisons with a mature Fortran code. Having developed, from scratch, Trixi.jl for a bit more than one year allows us to give an assessment of Julia for simulation-focused scientific computing. Based on our experience, we consider Julia to be suitable for simulation-focused scientific computing, in particular for hyperbolic PDEs, computational fluid dynamics, and related problems — at least on the scale of shared memory parallelism. The scalability of high-order methods for hyperbolic PDEs written in Julia to high-performance computing applications still needs to be demonstrated. Yet, we do not consider this an unsolvable problem, since Julia is not generically slower than traditional compiled HPC programming languages. Nevertheless, it appears to be more complicated to scale Julia to distributed systems without losing some of the simplicity and flexibility it offers for serial or shared-memory parallel computations.

Learning a new programming language naturally requires some time and effort. In our experience, however, it has paid off and we do not want to miss the many useful features of Trixi.jl enabled by Julia and its ecosystem. While there is no need to switch to Julia if people are satisfied with their existing tools, we encourage researchers to try out Julia for scientific computing and to stay for a while. The Julia community (on the Julia Discourse forum[24] or Slack/Zulip workspaces) is usually welcoming and helpful, both for new and experienced users.

[21]`https://github.com/Azrael3000/tmpi`

[22]`https://github.com/hpc/Spindle`

[23]`https://github.com/cea-hpc/wi4mpi`

Acknowledgments

We would like to thank all other contributors to and users of Trixi.jl for their help, encouragement, and discussions. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2044-390685587, Mathematics Münster: Dynamics-Geometry-Structure. Andrew Winters was funded through Vetenskapsrådet, Sweden grant agreement 2020-03642 VR. Jesse Chan acknowledges support from the US National Science Foundation under awards DMS-1719818 and DMS-1943186. Gregor J. Gassner and Michael Schlottke-Lakemper received funding from the European Research Council through the ERC Starting Grant "An Exascale aware and Un-crashable Space-Time-Adaptive Discontinuous Spectral Element Solver for Non-Linear Conservation Laws" (Extreme), ERC grant agreement no. 714487. The performance computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Tetralith partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

6. References

[1] Christoph Altmann. *Explicit discontinuous Galerkin methods for magnetohydrodynamics*. PhD thesis, University of Stuttgart, Germany, 2012. doi:10.18419/opus-3895.

[2] Santiago Badia and Francesc Verdugo. Gridap: An extensible finite element toolbox in Julia. *Journal of Open Source Software*, 5(52):2520, 2020. doi:10.21105/joss.02520.

[3] Nick Barnes. Publish your computer code: it is good enough. *Nature*, 467(7317), 2010. doi:10.1038/467753a.

[4] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018. doi:10.1109/TPDS.2018.2872064. arxiv:1712.03112 [cs.PL].

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671. arxiv:1411.1607 [cs.MS].

[6] Marvin Bohm, Andrew R. Winters, Gregor J. Gassner, Dominik Derigs, Florian Hindenlang, and Joachim Saur. An entropy stable nodal discontinuous Galerkin method for the resistive MHD equations. Part I: Theory and numerical verification. *Journal of Computational Physics*, 422:108076, 2020. doi:10.1016/j.jcp.2018.06.027.

[24]`https://discourse.julialang.org`

[7] Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011. doi:10.1137/100791634. `https://github.com/cburstedde/p4est`.

[8] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. MPI.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021. doi:10.21105/jcon.00068. `https://github.com/JuliaParallel/MPI.jl`.

[9] Jesse Chan, David C Del Rey Fernández, and Mark H Carpenter. Efficient entropy stable Gauss collocation methods. *SIAM Journal on Scientific Computing*, 41(5):A2938–A2966, 2019. doi:10.1137/18M1209234.

[10] Tianheng Chen and Chi-Wang Shu. Entropy stable high order discontinuous Galerkin methods with suitable quadrature rules for hyperbolic conservation laws. *Journal of Computational Physics*, 345:427–461, 2017. doi:10.1016/j.jcp.2017.05.025.

[11] Navid C Constantinou, Gregory LeClaire Wagner, Lia Siegelman, Brodie C Pearson, and André Palóczy. GeophysicalFlows.jl: Solvers for geophysical fluid dynamics problems in periodic domains on CPUs & GPUs. *Journal of Open Source Software*, 6(60):3053, 2021. doi:10.21105/joss.03053.

[12] David L Donoho. An invitation to reproducible computational research. *Biostatistics*, 11(3):385–388, 2010. doi:10.1093/biostatistics/kxq028.

[13] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[14] Roland Ewert and Wolfgang Schröder. Acoustic perturbation equations based on flow decomposition via source filtering. *Journal of Computational Physics*, 188(2):365–398, 2003. doi:10.1016/S0021-9991(03)00168-2.

[15] Travis C Fisher and Mark H Carpenter. High-order entropy stable finite difference schemes for nonlinear conservation laws: Finite domains. *Journal of Computational Physics*, 252:518–557, 2013. doi:10.1016/j.jcp.2013.06.014.

[16] Wolfgang Frings, Dong Ahn, Matthew LeGendre, Todd Gamblin, Bronis de Supinski, and Felix Wolf. Massively Parallel Loading. In *27th International Conference on Supercomputing, Eugene, OR, USA*, pages 389–398, 2013. doi:10.1145/2464996.2465020.

[17] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807623.

[18] Gregor Gassner. *Discontinuous Galerkin methods for the unsteady compressible Navier-Stokes equations*. PhD thesis, University of Stuttgart, Germany, 2009. doi:10.18419/opus-3788.

[19] Gregor J. Gassner, Florian Hindenlang, and Claus-Dieter Munz. A Runge-Kutta based discontinuous Galerkin method with time accurate local time stepping. *Adaptive High-Order Methods in Computational Fluid Dynamics*, 2:95–118, 2011. doi:10.1142/9789814313193_0004.

[20] Gregor J Gassner, Andrew R Winters, and David A Kopriva. Split form nodal discontinuous Galerkin schemes with summation-by-parts property for the compressible Euler equations. *Journal of Computational Physics*, 327:39–66, 2016. doi:10.1016/j.jcp.2016.09.013.

[21] T Grassi, S Bovino, DRG Schleicher, J Prieto, D Seifried, E Simoncini, and FA Gianturco. KROME – a package to embed chemistry in astrophysical simulations. *Monthly Notices of the Royal Astronomical Society*, 439(3):2386–2419, 2014. doi:10.1093/mnras/stu114.

[22] Jan S Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, volume 54 of *Texts in Applied Mathematics*. Springer Science & Business Media, New York, 2007. doi:10.1007/978-0-387-72067-8.

[23] Florian Hindenlang and Gregor J. Gassner. A new entropy conservative two-point flux for ideal MHD equations derived from first principles. Talk presented at HONOM 2019: European workshop on high order numerical methods, 2019.

[24] Sascha Hunold and Sebastian Steiner. Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC? In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 20–25, 2020. doi:10.1109/PMBS51919.2020.00008.

[25] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with Flux, 2018. arxiv:1811.01457 [cs.PL].

[26] Christopher A Kennedy and Mark H Carpenter. Fourth order 2N-storage Runge-Kutta schemes. Technical Memorandum NASA-TM-109112, NASA, NASA Langley Research Center, Hampton VA 23681-0001, United States, 1994.

[27] David A Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. Springer Science & Business Media, New York, 2009. doi:10.1007/978-90-481-2261-5.

[28] Nico Krais, Andrea Beck, Thomas Bolemann, Hannes Frank, David Flad, Gregor Gassner, Florian Hindenlang, Malte Hoffmann, Thomas Kuhn, Matthias Sonntag, and Claus-Dieter Munz. FLEXI: A high order discontinuous Galerkin framework for hyperbolic-parabolic conservation laws. *Computers & Mathematics with Applications*, 81:186–219, 2021. doi:10.1016/j.camwa.2020.05.004.

[29] Philippe G LeFloch, Jean-Marc Mercier, and Christian Rohde. Fully discrete, entropy conservative schemes of arbitrary order. *SIAM Journal on Numerical Analysis*, 40(5):1968–1992, 2002. doi:10.1137/S003614290240069X.

[30] Randall J LeVeque. Top ten reasons to not share your code (and why you should anyway). *SIAM News*, 46(3), 2013.

[31] Shengtai Li. An HLLC Riemann solver for magneto-hydrodynamics. *Journal of Computational Physics*, 203(1):344–357, 2005. doi:10.1016/j.jcp.2004.08.020.

[32] Yong Liu, Jianfang Lu, and Chi-Wang Shu. An oscillation-free discontinuous Galerkin method for hyperbolic systems. `https://www.brown.edu/research/projects/scientific-computing/sites/brown.edu.research.projects.scientific-computing/files/uploads/AN%20OSCILLATION-FREE%20DISCONTINUOUS%20GALERKIN%20METHOD%20FOR%20HYPERBOLIC%20SYSTEMS%20.pdf`, 2021.

[33] Sheehan Olver and Alex Townsend. A practical framework for infinite-dimensional linear algebra. In *2014 First Workshop for High Performance Technical Computing in Dynamic Languages*, pages 57–62. IEEE, 2014. doi:10.1109/HPTCDL.2014.10. Software available at `https://github.com/JuliaApproximation/ApproxFun.jl`.

[34] Samuel Omlin, Ludovic Räss, Grzegorz Kwasniewski, Benjamin Malvoisin, and Yury Podladchikov. Solving nonlinear multi-physics on GPU supercomputers with Julia. `https://www.youtube.com/watch?v=vPsfZUqI4_0`. Talk presented at JuliaCon, 2020.

[35] Avik Pal, Yingbo Ma, Viral Shah, and Christopher Rackauckas. Opening the blackbox: Accelerating neural differential equations by regularizing internal solver heuristics, 2021. arxiv:2105.03918 [cs.LG].

[36] Matteo Parsani, Radouan Boukharfane, Irving Reyna Nolasco, David C Del Rey Fernández, Stefano Zampini, Bilel Hadri, and Lisandro Dalcin. High-order accurate entropy-stable discontinuous collocated Galerkin methods with the summation-by-parts property for compressible CFD frameworks: Scalable SSDC algorithms and flow solver. *Journal of Computational Physics*, 424:109844, 2021. doi:10.1016/j.jcp.2020.109844.

[37] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, 5(1):15, 2017. doi:10.5334/jors.151.

[38] Ali Ramadhan, Gregory LeClaire Wagner, Chris Hill, Jean-Michel Campin, Valentin Churavy, Tim Besard, Andre Souza, Alan Edelman, Raffaele Ferrari, and John Marshall. Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *Journal of Open Source Software*, 5(53):2018, 2020. doi:10.21105/joss.02018.

[39] Hendrik Ranocha. Comparison of some entropy conservative numerical fluxes for the Euler equations. *Journal of Scientific Computing*, 76(1):216–242, 2018. doi:10.1007/s10915-017-0618-1. arxiv:1701.02264 [math.NA].

[40] Hendrik Ranocha. *Generalised Summation-by-Parts Operators and Entropy Stability of Numerical Methods for Hyperbolic Balance Laws*. PhD thesis, TU Braunschweig, Germany, 2018.

[41] Hendrik Ranocha. Entropy conserving and kinetic energy preserving numerical methods for the Euler equations using summation-by-parts operators. In Spencer J. Sherwin, David Moxey, Joaquim Peiró, Peter E. Vincent, and Christoph Schwab, editors, *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2018*, pages 525–535, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-39647-3_42.

[42] Hendrik Ranocha. Optimizing Julia code: Improving the performance of entropy-conservative DG methods in Trixi.jl. `https://ranocha.de/blog/Optimizing_EC_Trixi`, 2021.

[43] Hendrik Ranocha. SummationByPartsOperators.jl: A Julia library of provably stable semidiscretization techniques with mimetic properties. *Journal of Open Source Software*, 6(64):3454, 08 2021. doi:10.21105/joss.03454.

[44] Hendrik Ranocha, Lisandro Dalcin, Matteo Parsani, and David I. Ketcheson. Optimized Runge-Kutta methods with automatic step size control for compressible computational fluid dynamics. *Communications on Applied Mathematics and Computation*, 11 2021. doi:10.1007/s42967-021-00159-w. arxiv:2104.06836 [math.NA].

[45] Hendrik Ranocha and Gregor J Gassner. Preventing pressure oscillations does not fix local linear stability issues of entropy-based split-form high-order schemes. *Communications on Applied Mathematics and Computation*, 2021. doi:10.1007/s42967-021-00148-z. arxiv:2009.13139 [math.NA].

[46] Hendrik Ranocha, Michael Schlottke-Lakemper, Jesse Chan, Andrés M Rueda-Ramírez, Andrew R Winters, Florian Hindenlang, and Gregor J Gassner. Efficient implementation of modern entropy stable and kinetic energy preserving discontinuous Galerkin methods for conservation laws, 12 2021. arxiv:2112.10517 [cs.MS].

[47] Hendrik Ranocha, Michael Schlottke-Lakemper, Andrew Ross Winters, Erik Faulhaber, Jesse Chan, and Gregor Gassner. Reproducibility repository for adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing. `https://github.com/trixi-framework/paper-2021-juliacon`, 2021. doi:10.5281/zenodo.5201484.

[48] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia, 2016. arxiv:1607.07892 [cs.MS].

[49] Michael Schlottke-Lakemper, Andrew R Winters, Hendrik Ranocha, and Gregor J Gassner. A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics. *Journal of Computational Physics*, page 110467, 2021. doi:10.1016/j.jcp.2021.110467. arxiv:2008.10593 [math.NA].

[50] Vikram Singh and Praveen Chandrashekar. On a linear stability issue of split form schemes for compressible flows, 2021. arxiv:2104.14941 [math.NA].

[51] Eitan Tadmor. The numerical viscosity of entropy stable schemes for systems of conservation laws. I. *Mathematics of Computation*, 49(179):91–103, 1987. doi:10.1090/S0025-5718-1987-0890255-3.

[52] Eleuterio F Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer, Berlin Heidelberg, 2009. doi:10.1007/b79761.

APPENDIX

Julia does not perform automatic fused multiply-add (FMA) contraction, i.e., replacing a multiplication followed by an addition with a single FMA instruction, as it is inconsistent with its strict floating point semantics. In other compilers, this optimization can be controlled by a global compiler option (`-ffp-contract` in GCC and Clang) and is often enabled at reasonable optimization levels such as `-O2` for GCC and Intel compilers[25]. In Julia, FMA contraction is controlled more locally: the `muladd` intrinsic function allows the compiler to evaluate a multiply-add using the most efficient method. Relying on Julia's code manipulation techniques, the `@muladd` macro makes it convenient to use the `muladd` function by syntactically rewriting expressions to insert `muladd` in appropriate locations.

[25]An interesting coincidence is that not only Julia but most LLVM-based frontends tend not to enable FMA contraction by default.