DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of a mixed-precision finite volume limiter for the ADER-DG algorithm

Felix Zhengbuo Guan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of a mixed-precision finite volume limiter for the ADER-DG algorithm

Evaluierung eines gemischten Präzisions-Finite-Volumen-Limiters für den ADER-DG-Algorithmus

Author:	Felix Zhengbuo Guan
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M.Sc. Marc Marot-Lassauzaie
Submission Date:	01.08.2025

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 01.08.2025 Felix Zhengbuo Guan

Acknowledgments

I would like to express my sincere gratitude to everyone who supported me throughout the course of my bachelor thesis. First and foremost, I want to thank my supervisor, Marc, for their valuable guidance, insightful feedback, and continuous encouragement. A heartfelt thanks to my family and friends for their unwavering support and motivation during this work. This achievement would not have been possible without them.

Abstract

This thesis presents a mixed-precision implementation of a finite volume limiter for the ADER-DG Algorithm. The implementation extends the Python API of the ExaHyPE engine, which is built on top of the Peano framework. It allows for specifying the numerical precision of the individual kernels of the ADER solver and Finite Volume Limiter, and is then used to study the resulting error margin when using mixed precision. In particular, we assess how sensitive the kernels of the finite volume limiter are to the use of double, single, and half precision. The question addressed is, in which settings mixed precision is viable, considering its potential error margin.

Contents

1 Introduction

High-performance computing (HPC) applications are increasingly relying on numerical methods that balance accuracy and computational efficiency. One such method is the Arbitrary DERivative Discontinuous Galerkin (ADER-DG) algorithm. A crucial component of the algorithm is the finite volume limiter, which tries to ensure numerical stability when dealing with high-order gradients and discontinuities.

With the release of support for two different 16-bit floating point types (see table 1.1) in the newest C++23 standard [Int24] and the growing importance of hardware accelerators such as GPUs, it is clear that mixed-precision computing has emerged as a promising technique. Mixed-precision and especially exploiting low precision can improve performance due to reduced memory usage, lower costs of data transfer, higher cache locality, and higher intra-instruction level parallelism. This can, for example, be seen in the works of Gupta et al.[Gup+15] and Micikevicius et al. [Mic+17], where low-precision types were used to train neural networks. The challenge lies in adaptively choosing the precision in different algorithms and implementations while maintaining acceptable levels of numerical accuracy.

This thesis explores a mixed-precision implementation of the finite volume limiter within the ADER-DG algorithm. We examine both the ADER-DG algorithm with a posteriori subcell limiting introduced by Dumbser et al. [Dum+14], as well as the ADER-DG algorithm with static subcell limiting. An implementation of both is supported in the hyperbolic partial differential equation (PDE) engine ExaHyPE [Rei+20]. The ExaHyPE engine provides a generic implementation of the ADER-DG algorithm and allows the user to specify problem specifics and the respective limiter through a Python API. The presented mixed-precision implementation simply extends this Python API to enable the configuration of the precisions of each major kernel of the limiters.

A similar study by Marot-Lassauzaie et al. [MB25], has already investigated a mixed-precision implementation for the ADER-DG algorithm implemented in ExaHyPE. They enable users to configure precision individually for the core kernels of the ADER-DG algorithm and then assess how mixed precision arithmetic affects convergence and stability across four representative hyperbolic PDEs (elastic/acoustic waves, shallow water, Euler equations). Their key finding is that while fp64 is necessary for high-order convergence, fp32 may suffice in moderate-resolution cases, and pure fp16 or bf16 often fails—though mixed-precision configurations can sometimes restore stability.

However, their investigation explicitly excludes scenarios involving the finite volume limiter, focusing solely on smooth problems where no limiting is required. This leaves open the question of how mixed-precision strategies perform in the presence of discontinuities and shocks. This is precisely where the limiter becomes active and numerical robustness is critical.

Even though performance and memory usage are two key motivations for using low or mixed precision, we only focus on the accuracy of the algorithm. This is because performance and memory usage are strongly dependent on the underlying hardware. Especially when using the half-precision types float16 and bfloat16, performance and memory usage can deteriorate severely if the hardware does not support them.

In chapter 2 we give an abstract introduction to the underlying theoretical backgrounds of the algorithms. Chapter 3 briefly explains the framework and how the algorithms' components are employed. Finally, chapter 4 visualizes the impact of mixed precision, and chapter 5 gives a summary of which practices are reasonable when employing mixed-precision.

Type	**Standard**	**Mantissa bits**	**Exponent bits**
`bfloat16_t`	Brain floating point	7	8
`float16_t`	IEEE binary 16	10	5
`float32`	IEEE binary 16	23	8
`float64`	IEEE binary 16	52	11

Table 1.1: C++23 floating-point types

2 Theoretical Background

This chapter introduces some of the theoretical backgrounds that are helpful to understand the project as a whole. Since mathematical detail is not crucial to our argument, we will refrain from exploring it here.

2.1 The ADER-DG Algorithm

The ADER-DG algorithm combines Arbitrary DERivative (ADER) time integration with the Discontinuous Galerkin (DG) method to solve hyperbolic partial differential equations with high-order accuracy in both space and time. The computational domain is divided into a mesh of elements (or cells), where each element holds a high-order polynomial approximation of the solution. These polynomials are allowed to be discontinuous across cell boundaries, which gives the method flexibility and makes it ideal for adaptive mesh refinement and complex geometries. Instead of using traditional time-stepping like Runge-Kutta [GMM15], ADER-DG computes the solution over an entire time step using one local space-time prediction within each cell. This one-step update significantly reduces the need for synchronization across domain boundaries, which enhances parallel scalability and overall efficiency. To account for interactions between cells, numerical fluxes are computed at the cell interfaces. While this works well in smooth regions, non-physical oscillations can occur near shocks or discontinuities. To prevent this, ADER-DG is often combined with a finite-volume subcell limiter that locally switches to a more robust scheme where needed. This combination of accuracy, efficiency, and robustness makes ADER-DG a powerful tool for large-scale simulations in fields like fluid dynamics, astrophysics, and seismology. For a detailed explanation of ADER and DG, we refer the reader to the works by Titarev et al. [TT02] and Shu [Shu09], respectively.

2.2 The Finite Volume Limiter

As already mentioned, the ADER-DG method delivers high-order accuracy in regions where the solution is smooth. However, near discontinuities, such as shocks or steep gradients, it can produce oscillations that lead to physically inadmissible values. To

maintain both accuracy and stability, a finite volume limiter is incorporated. The core idea is to identify areas in the computational domain—called troubled cells—where the ADER-DG solution becomes unreliable. Instead of applying the high-order DG method there, the solution is locally recomputed using a more robust finite volume scheme. This finite volume approach is inherently more stable near discontinuities and effectively controls oscillations by using piecewise constant approximations instead of high-degree polynomials. After performing a stable update on the subcells, the limited solution is projected back into a high-order polynomial representation, allowing the method to preserve its overall accuracy where the solution remains smooth. By combining the precision of ADER-DG in smooth regions with the robustness of finite volume methods near discontinuities, the finite volume limiter ensures that the solver is both accurate and stable across a wide range of challenging problems.

2.2.1 A Posteriori Limiting

One way to employ the finite volume limiter within the ADER-DG algorithm is through an a posteriori limiting approach. This technique involves detecting troubled cells **after** computing the high-order discontinuous Galerkin solution and then locally applying the more robust finite volume scheme to maintain stability and prevent oscillations near discontinuities.

Physical Admissibility Criterion

One method for detecting troubled cells is the Physical Admissibility Criterion. As the name suggests, this method works by checking whether certain values are violating physical properties. An example would be that the density or water level inside a cell should not be negative. Users have the ability to define these properties, which will then be validated. If a property is violated in a cell, the cell is marked as troubled, and the finite volume scheme is triggered.

Discrete Maximum Principle

Another method for detecting troubled cells is the Discrete Maximum Principle. This method uses information from neighboring cells to determine if the current cell is troubled. To do so, an interval based on the minimum and maximum values of the neighboring cells is defined. If the solution of the current cell after the next step now overshoots or undershoots this interval, the cell is marked as troubled. Although this method does not guarantee physical admissibility, it can still be used to maintain numerical stability. The idea behind this criterion is also related to smoothness: If a cell violates the discrete maximum principle, for example by being smaller than both its

left and right neighbors, it creates a local minimum, implying a gradient change from negative to positive within a single cell. This kind of behavior signals a kink in the current cell and is precisely what we aim to detect and correct.

A Posteriori Limiting is useful when it is difficult to foresee where physically inadmissible values might develop. The downside of this flexibility is that this method can be computationally expensive. Checking for errors requires communication between neighboring regions, which is expensive, especially for parallelism. Also, a rollback and recomputation of the solutions is needed, which in ExaHyPE requires multiple additional traversals over the domain. More on A Posteriori Limiting can be viewed in the work by Zanotti et al. [Zan+15].

2.2.2 Static Limiting

Another way of employing the finite volume limiter is through static limiting. While a posteriori limiting dynamically applies the finite volume limiter only when certain conditions are violated during the simulation, static limiting takes a more conservative approach. In static limiting, the decision to apply a limiter is made in advance, typically based on fixed geometric, physical, or user-defined indicators that mark specific regions of the domain as "sensitive". This approach reduces the overhead associated with a posteriori limiting by allowing communication to be established once at the beginning, since it remains unchanged over time. However, this approach requires insight into the "sensitive" regions of the problem, as areas outside the scope of the static limiter will not benefit from any limiting.

3 Implementation Details

This chapter first outlines the technical structure of the project, describing how various tools such as Peano, ExaHyPE, Jinja2, and Python interact to create a user-defined simulation executable. Secondly, we will also show the changes that were necessary to realize a mixed precision finite volume limiter.

3.1 Framework Introduction

3.1.1 Peano

At the heart of the project lies the Peano framework, a highly efficient, Adaptive Mesh Refinement (AMR) engine designed for dynamically adaptive cartesian grids. Peano stores these meshes in a spacetree structure and relies on space-filling curves (such as the Peano or Hilbert curve) to efficiently traverse these. This enhances cache locality and supports parallelism, making it suitable for many high-performance computing applications. To read more on the Peano Framework, we refer to the work of Weinzierl [Wei19].

3.1.2 ExaHyPE

Building on top of Peano, ExaHyPE provides a lightweight layer for defining and solving hyperbolic partial differential equations (PDEs). ExaHyPE delegates grid traversal and task orchestration to Peano, while exposing a flexible interface for problem-specific physics and solvers. Notably, ExaHyPE allows the user to define problem specifics such as solvers, fluxes, initial conditions, and boundary conditions. For a comprehensive overview of the flexibility offered by ExaHyPE, we refer the reader to the work of Reinarz et al. [Rei+20]. The general form of PDEs solved by ExaHyPE is given by:

$$\frac{\partial Q}{\partial t} + \nabla \cdot \mathbf{F}(Q, \nabla Q) + \mathbf{B}(Q) \cdot \nabla Q = \mathbf{S}(Q) + \sum_{i=1}^{n_\psi} \delta_i \tag{3.1}$$

where

- $\mathbf{Q} = (Q_1, Q_2, \ldots, Q_m)^T$ is the vector of conserved variables (e.g., water height, density).

- $\frac{\partial \mathbf{Q}}{\partial t}$ denotes the time derivative of the conserved variables.
- $\nabla \cdot \mathbf{F}(\mathbf{Q}, \nabla \mathbf{Q})$ is the divergence of the flux tensor, which may depend on both $\mathbf{Q}$ and its spatial gradients $\nabla \mathbf{Q}$. This term represents conservative fluxes.
- $\mathbf{B}(\mathbf{Q}) \cdot \nabla \mathbf{Q}$ represents non-conservative products, which cannot be expressed as a divergence of a flux and may model geometric or physical effects.
- $\mathbf{S}(\mathbf{Q})$ includes source terms depending on the state vector only, such as gravitational forces or friction.
- $\sum_{i=1}^{n_\psi} \delta_i$ are distributional source terms (e.g., Dirac delta functions) that model localized singular sources.

Through utilizing the backbone of Peano and by supporting the flexible definition of problem specifics through ExaHyPE, it is possible to build a simulation for a hyperbolic PDE with relatively low effort while still maintaining high computational efficiency and scalability.

3.1.3 Python API

As mentioned earlier, ExaHyPE provides a flexible interface to define problem specifics. This is realized by a high-level Python API that acts as the front-end interface for defining simulation configurations. The problem specifics, such as solvers, fluxes, initial conditions, and boundary conditions, are parsed by a function in the Python script to define a dictionary that maps a variable to the individual parameters. To illustrate this process, an example of parsing the flux is shown in Figure 3.1. Later, these variables can be used to dynamically populate a set of templated C++ source files with the problem-specific data.

(a) Function definition and call

```
def set_implementation(self, flux):
    self._flux_implementation = flux

aderdg_solver.set_implementation(
    flux="""
        double ih = 1.0 / Q[0];

        F[0] = Q[1 + normal];
        F[1] = Q[1 + normal] * Q[1] * ih;
        F[2] = Q[1 + normal] * Q[2] * ih;
        F[3] = 0.0;
    """
)
```

(b) Dictionary entry

```
def _init_dictionary_with_default_parameters(self, d):
    d["FLUX_IMPLEMENTATION"] = self._flux_implementation
```

Figure 3.1: Example: parsing the flux for the ADER-DG solver

3.1.4 C++ Templates

The computational core of the simulations is implemented using C++ templates, enabling the injection of problem-specific components and configuration parameters at compile time. Templates allow static polymorphism, eliminating runtime overhead and enabling compiler optimizations such as inlining and loop unrolling. This is done by utilizing the Jinja2 engine [Ron08], which injects the symbolic expressions (e.g., fluxes, boundary conditions) and configuration parameters (e.g., precision) according to the dictionary defined in the Python layer. Figure 3.2 shows how the flux is being injected into the C++ code through the dictionary.

(a) Jinja2 template

```
"""
static inline GPUCallableMethod
    void flux(
      ...
    ) InlineMethod {
      {% if FLUX_IMPLEMENTATION!="<empty>" %}
      {{FLUX_IMPLEMENTATION}}
      {% endif %}
    }
"""
```

(b) Resulting C++ function

```
static inline GPUCallableMethod
    void flux(
      ...
    ) InlineMethod {
        double ih = 1.0 / Q[0];

        F[0] = Q[1 + normal];
        F[1] = Q[1 + normal] * Q[1] * ih;
        F[2] = Q[1 + normal] * Q[2] * ih;
        F[3] = 0.0;
    }
```

Figure 3.2: Example: injecting the flux into C++ code

3.1.5 Code Generation

The core non-problem-specific components of Peano and ExaHyPE are precompiled. Once the problem-specific C++ code is populated, it is compiled and linked with the Peano and ExaHyPE backends to produce a standalone simulation executable. This closes the loop between user-level problem specification and high-performance numerical execution.

3.2 The Limiting Kernel

The ADER-DG solver and the finite volume solver are already able to support different precision types for their kernels. For this work, only the implementation of the limiting kernel had to be adjusted. The limiting kernel handles identifying troubled cells and mapping the solution between the DG space and the finite volume space. When a cell is marked as troubled, the solution is recomputed based on the solutions of the previous step with the finite volume solver. For this, a mapping between the DG space and the finite volume space is necessary, as they use fundamentally different representations of the solution inside the cell. The DG method represents the solution inside each cell as a Lagrange polynomial, which is stored implicitly as a matrix containing the value of the polynomial at each nodal support point. The finite volume method in contrast, stores the solution as cell averages of a finer subcell.

3.2.1 Adjusting the Python API

As introduced in chapter 2.2.1 and 2.2.2, the limiter has two different implementations. For both implementations, a very similar Python API exists. Both of them were extended by a function and a dictionary entry as seen in figure 3.3. Notice that the precision of the regular solver and the limiting solver, in our case the ADER-DG and finite volume solver are also being parsed as they will be needed to map the solutions.

(a) Python function

```
def set_limiter_precisions(self, precision):
        self._precision = PrecisionType[precision]
        self.create_data_structures()
        self.create_action_sets()
```

(b) Dictionary entries

```
d["REGULAR_SOLVER_STORAGE_PRECISION"] = self._regular_solver._precision
d["LIMITER_SOLVER_STORAGE_PRECISION"] = self._limiter_solver._precision
d["LIMITER_PRECISION"] = self._precision
```

Figure 3.3: Parsing the dictionary of the limiter

3.2.2 Adjusting the C++ Templates

To adjust the C++ templates, we now replace the hardcoded precision types with the corresponding variable of the dictionary. An example function for the use case of identifying troubled cells and mapping the solution between the DG space and the finite volume space can be seen in figure 3.4. Here, it is important that the function parameters related to the solution spaces of the different solvers use the correct corresponding precision types.

(a) Function mapping solutions

```
void projectOnDGSpaceFromFVWithoutHalo(
  const {{LIMITER_SOLVER_STORAGE_PRECISION}}* const lim,
  {{REGULAR_SOLVER_STORAGE_PRECISION}}* const luh
) {
    ... //maps from finite volume space onto DG space
}
```

(b) Function identifying troubled cells

```
bool discreteMaximumPrincipleAndMinAndMaxSearch(
  {{FULL_QUALIFIED_SOLVER_NAME}}& solver,
  const {{REGULAR_SOLVER_STORAGE_PRECISION}}* const luh,
  const {{LIMITER_PRECISION}} relaxationParameter,
  const {{LIMITER_PRECISION}} differenceScaling,
  {{LIMITER_PRECISION}}* boundaryMinPerObservable,
  {{LIMITER_PRECISION}}* boundaryMaxPerObservable
) {
    ... //marks cells as troubled with the discrete maximum principle
}
```

Figure 3.4: Example functions for solution mapping and identifying troubled cells

4 Results

This chapter discusses the impact of mixed-precision on the finite volume limiter.

4.1 Scenarios

To evaluate the different mixed-precision implementations, we apply them to four different simulations. This section first introduces these simulations and gives a point of reference to the mixed-precision implementations. All simulations are conducted in two dimensions, though similar behavior is expected to occur in three dimensions as well.

4.1.1 Shock Tube

This problem represents a two-dimensional extension of the classical Sod shock tube problem. These problems simulate the change of density of a fluid using the compressible Euler equations. Initially, the domain is split into two parts. The lower left quadrant is initialized with low density and pressure, and the rest of the domain is initialized with high density and pressure. This sets up a shock that evolves into a complex wave structure, including further shocks and discontinuities, over time. The vector of conserved variables is defined as:

$$\mathbf{Q} = \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ E_t \end{pmatrix} \tag{4.1}$$

where

- ρ is the fluid density,
- ρu_1 and ρu_2 are the momenta in the x- and y-directions respectively,
- E_t is the total energy.

To define the described initial state, the initial values for the conserved variables are set as:

$$\rho = \begin{cases} 0.125, & \text{if } x < -x_s \text{ and } y < -x_s, \\ 1.0, & \text{otherwise,} \end{cases} \quad \rho u_1 = 0.0, \quad \rho u_2 = 0.0, \quad E = \frac{p}{\gamma - 1},$$

with

$$p = \begin{cases} 0.1, & \text{if } x < -x_s \text{ and } y < -x_s, \\ 1.0, & \text{otherwise,} \end{cases}$$

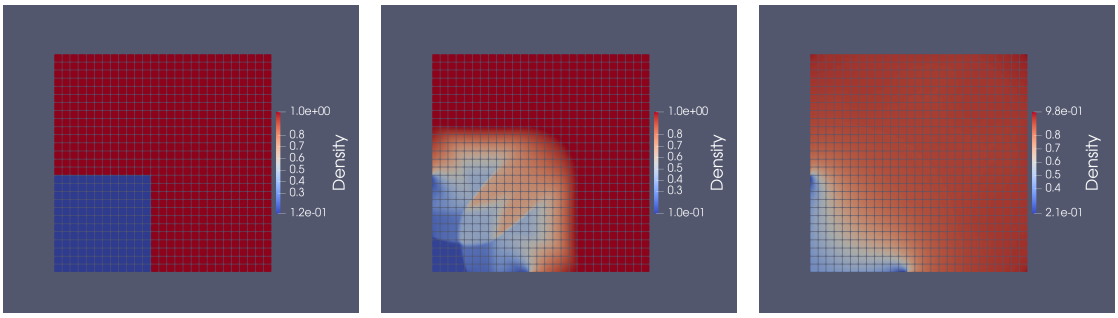
Here p is the pressure, γ is the adiabatic index (e.g., 1.4 for air), and x_s is the variable defining the border. The resulting PDE is of the form:

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ E_t \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho u_1 \\ \rho u_1^2 + p \\ \rho u_2 u_1 \\ (E_t + p) u_1 \end{pmatrix} + \begin{pmatrix} \rho u_2 \\ \rho u_1 u_2 \\ \rho u_2^2 + p \\ (E_t + p) u_2 \end{pmatrix} = \mathbf{0} \tag{4.2}$$

Notice that the right-hand side of the equation is zero, because the source terms $\mathbf{S}(\mathbf{Q})$are defined as 0. This means that there are no additional external influences or forces. The eigenvalues, which represent the speeds at which different types of waves propagate through the compressible fluid, are given by:

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} u - c \\ u \\ u + c \end{pmatrix}, \tag{4.3}$$

where u is the fluid velocity in the direction of wave propagation, and c is the wave propagation speed, defined by $c = \sqrt{\frac{\gamma p}{\rho}}$. The problem uses a posteriori limiting as the troubled cells are expected to follow the "shock front" that moves outward across the domain. A polynomial order of 5 is used in the ADER-DG solver. The cell size is consistent between the ADER-DG solver and the finite volume limiter, with each cell measuring approximately 0.0815 in both length and width. A visualization of this problem can be seen in figure 4.1.



(a) Start (b) Mid (c) End

Figure 4.1: Simulation stages of the Shock Tube Simulation: start, mid, and end.

4.1.2 Naca Airfoil

This simulation models the flow around a symmetric NACA airfoil using the same compressible Euler equations in two dimensions as the Shock Tube simulation (see 4.1.1). This NACA airfoil is defined by the equation for a symmetrical NACA airfoil, which specifies the thickness distribution as:

$$y_t(x) = 5tc\left(0.2969\sqrt{\frac{x}{c}} - 0.1260\frac{x}{c} - 0.3516\left(\frac{x}{c}\right)^2 + 0.2843\left(\frac{x}{c}\right)^3 - 0.1015\left(\frac{x}{c}\right)^4\right), \tag{4.4}$$

where

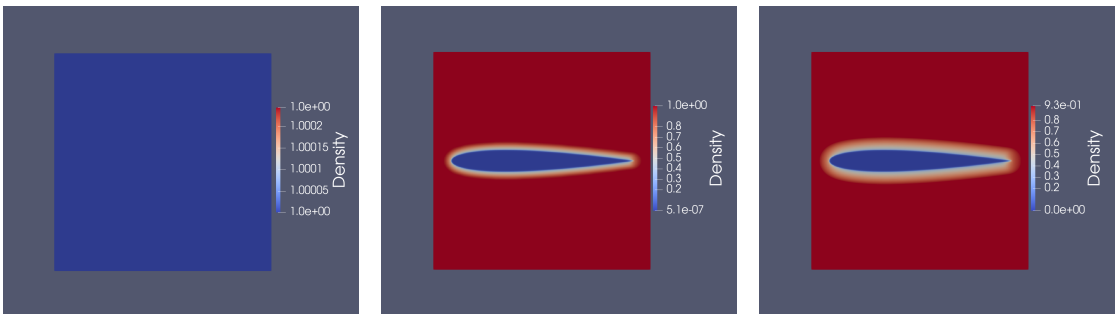
- c is the chord length of the airfoil, i.e., the distance from the leading edge to the trailing edge,
- x is the position along the chord from 0 to 1 (0 to 100%),
- t is the maximum thickness as a fraction of the chord length,
- $y_t(x)$ is the half-thickness of the airfoil at a given value of x.

In this specific case we use $c = 100$ and $t = 0.12$. The domain is a square region centered at the airfoil, with flow conditions initialized to represent uniform upstream flow from left to right. The vector of conserved variables is the same as defined in equation 4.1. The initial conditions specify a uniform flow with:

$$\rho = 1.0, \quad \rho u_1 = 0.1, \quad \rho u_2 = 0.0, \quad E_t = 1.0,$$

To model the airfoil as a solid obstacle within the domain, it is implicitly represented via an auxiliary variable, which indicates the proportion of the cell that is not part

of the airfoil. To handle fluxes at the airfoil boundary, the Riemann solver employed reflects the ingoing flux proportionally to the unavailable volume fraction in a cell. Because this approach is not physically consistent for ADER-DG, cells with a volume fraction less than 1 are statically limited by the finite volume limiter. The definition of the PDE as well as the eigenvalues are also the same as seen in equation 4.2 and 4.3 respectively. The ADER-DG solver operates with a polynomial order of 5, and both it and the finite volume limiter use cells that are about 1.63 units long and wide. A visualization of the simulation is provided in figure 4.2.



(a) Start (b) Mid (c) End

Figure 4.2: Simulation stages of the NACA airfoil simulation: start, mid, and end.

4.1.3 Shallow Water

This problem models the evolution of a circular "dam break" using the two-dimensional shallow water equations. The domain is initialized with a radially symmetric, smoothly varying water surface that is highest at the center and decreases gradually outward. This setup mimics the sudden collapse of a raised water column, generating radial waves that propagate outward. The water is shallow as the initial vertical difference between the highest and lowest point of the water is proportionally small in comparison to its horizontal extent. The boundary is defined to copy the value inside the boundary to the outside, which allows waves to "leave" the domain. Therefore, the simulation is expected to evolve toward an equilibrium state. In this case, the vector of conserved variables can be defined as:

$$\mathbf{Q} = \begin{pmatrix} h \\ hu_1 \\ hu_2 \\ b \end{pmatrix}, \tag{4.5}$$

where

- h is the water height,

- hu_1 and hu_2 are the momenta in the x- and y-directions respectively,
- b is the bathymetry or bottom topography.

To define the initial state, the initial values for these variables are set as:

$$h = 4.0, \quad hu_1 = 0.0, \quad hu_2 = 0.0, \quad b(x) = 2.0 - \|\mathbf{x}\|_2,$$

Here $\mathbf{x}$ is the distance to the domain center. Following this, the PDE can be written as:

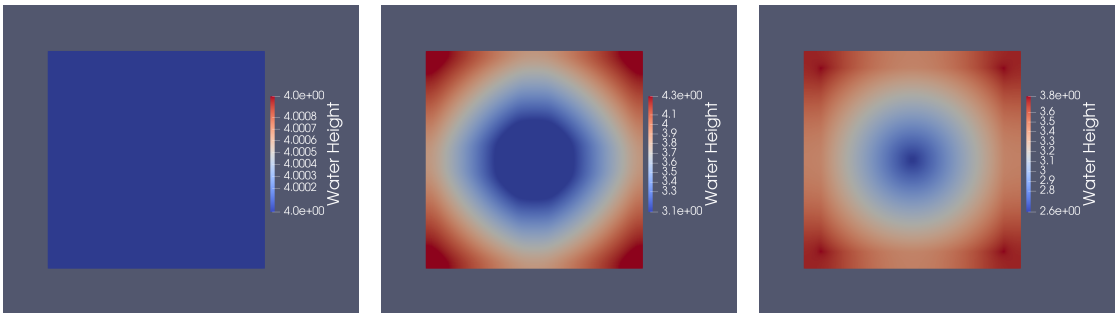
$$\frac{\partial}{\partial t}\begin{pmatrix} h \\ hu_1 \\ hu_2 \\ b \end{pmatrix} + \nabla \cdot \begin{pmatrix} hu_1 & hu_2 \\ hu_1^2 & hu_1u_2 \\ hu_1u_2 & hu_2^2 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ hg\frac{\partial}{\partial x}(b+h) \\ hg\frac{\partial}{\partial y}(b+h) \\ 0 \end{pmatrix} = \mathbf{0}. \tag{4.6}$$

where $g = 9.81\,\mathrm{m/s^2}$ is the gravitational acceleration constant. Same as in the 2D Shock Tube scenario 4.1.1, there are no additional external influences resulting in the right-hand side of the equation being zero. The eigenvalues of the system, which characterize its wave speeds, are given by:

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} u + \sqrt{g(h+b)} \\ u \\ u - \sqrt{g(h+b)} \end{pmatrix}, \tag{4.7}$$

where u represents the velocity component normal to the wave propagation direction.

Due to the radial symmetry and smooth initial condition, this problem also tests the scheme's ability to preserve symmetry and resolve smooth wave propagation without introducing oscillations. A static limiter is used at the boundaries of the domain, as using solely ADER-DG would create oscillations near them. This is caused by the implementation of the boundary condition. The ADER-DG solver employs a polynomial order of 5. Both the ADER-DG solver and the finite volume limiter use cells that are approximately 0.0815 units in length and width. A visualization of this simulation is provided in figure 4.3.



(a) Start (b) Mid (c) End

Figure 4.3: Simulation stages of the Shallow Water Simulation: start, mid, and end.

4.1.4 Landslide

This simulation models a granular landslide using a variation of the two-dimensional shallow water equations, which approximate granular flow with a free surface under the influence of gravity. The setup involves an inclined terrain where a circular column of dense material is released, propagating from high (left) to low (right) and spreading over time. The conserved variables are the same as in equation 4.5. The initial state defines a circular column of height $h_0 = 0.008$ centered at $(x_0, y_0) = (0.15, 0.79)$ with radius $r_0 = 0.10$, and zero initial momentum. The terrain is sloped at an angle $\zeta = 35°$, decreasing linearly in the x-direction. This initial state is given by these initial values:

$$h(\mathbf{x}, 0) = \begin{cases} 0.008, & \text{if } \|\mathbf{x} - \mathbf{x}_0\|_2 < 0.10, \\ 0, & \text{otherwise,} \end{cases} \quad hu_1 = 0.0, \quad hu_2 = 0.0, \quad b(x) = (L_x - x)\tan(\zeta),$$

with $\mathbf{x}_0 = (0.15, 0.79)$ being the center of the circular column and $L_x = 1.58$ being the length of the domain in the x-direction. The PDE for this problem can be written as:

$$\frac{\partial}{\partial t}\begin{pmatrix} h \\ hu_1 \\ hu_2 \end{pmatrix} + \nabla \cdot \begin{pmatrix} hu_1 & hu_2 \\ hu_1^2 + \frac{1}{2}gh^2\cos\zeta & hu_1u_2 \\ hu_1u_2 & hu_2^2 + \frac{1}{2}gh^2\cos\zeta \end{pmatrix} = \begin{pmatrix} 0 \\ ghS_x \\ ghS_y \end{pmatrix} + \nabla \cdot \begin{pmatrix} 0 & 0 \\ D_{1,x} & D_{1,y} \\ D_{2,x} & D_{2,y} \end{pmatrix} \tag{4.8}$$

where the source terms S_x and S_y account for gravitational and frictional effects, given by:

$$S_x = \cos\zeta\left(\tan\zeta - \mu\frac{u_1}{|\bar{\mathbf{u}}|}\right), \quad S_y = -\mu\cos\zeta\frac{u_2}{|\bar{\mathbf{u}}|} \tag{4.9}$$

and the diffusive terms are:

$$D_{1,x} = \nu h^{3/2} \frac{\partial u_1}{\partial x}, \tag{4.10}$$

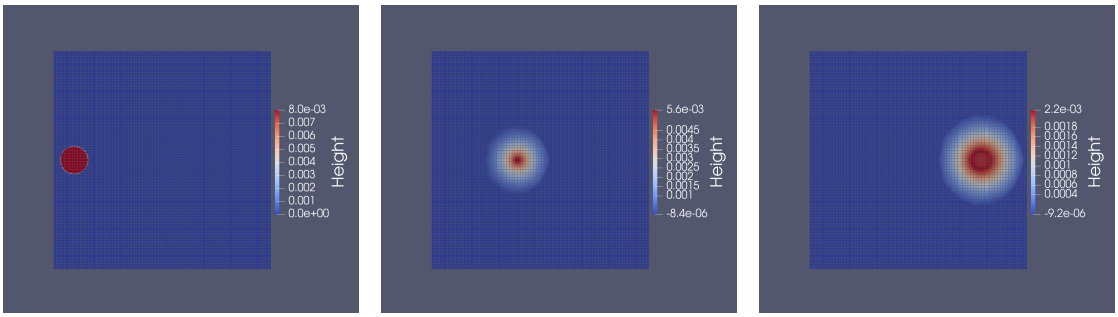
$$D_{1,y} = D_{2,x} = \frac{1}{2} \nu h^{3/2} \left(\frac{\partial u_1}{\partial y} + \frac{\partial u_2}{\partial x} \right), \tag{4.11}$$

$$D_{2,y} = \nu h^{3/2} \frac{\partial u_2}{\partial y} \tag{4.12}$$

Here,

1. $|\bar{\mathbf{u}}| = (u_1, u_2)$ is the depth-averaged velocity,
2. $g = 9.81\,\mathrm{m/s^2}$ is the gravitational acceleration,
3. μ is the basal friction coefficient,
4. ν is a viscosity-like parameter.

The diffusive terms represent the internal friction and viscous-like effects within the flowing material. They act to smooth velocity gradients by modeling momentum diffusion, which helps stabilize the numerical solution and captures physical processes such as shear stresses and turbulence effects. In particular, these terms introduce a form of viscosity parameterized by ν, which controls the intensity of the momentum diffusion based on the flow thickness h and velocity gradients. Without these terms, the model would only describe idealized inviscid flow and could also lead to numerical instabilities. The bathymetry modelled by $b(x)$ is implicitly included by the source terms. For this scenario, a posteriori limiting is used as the moving granular material causes the shocks. The ADER-DG solver uses a polynomial order of 4. Both the ADER-DG solver and the finite volume limiter use the same cell size, which is approximately 0.215 long and wide. A visualization of this simulation can be seen in 4.4.



(a) Start (b) Mid (c) End

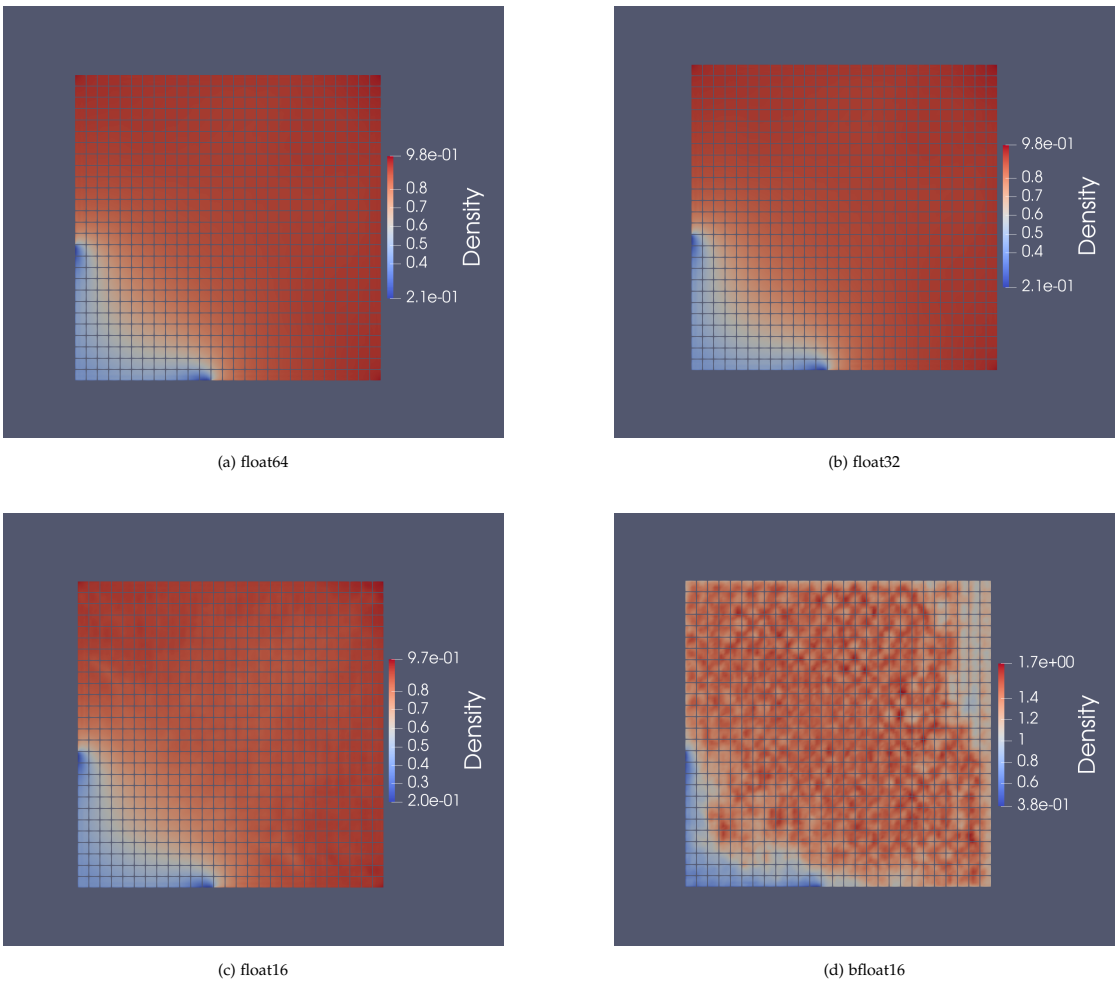
Figure 4.4: Simulation stages of the Landslide Simulation: start, mid, and end.

4.2 Mixed-Precision Implementations

The mixed-precision implementations discussed here use uniform double precision for the ADER-DG component and uniform double, single, and half precision for the finite volume limiter. For a broader discussion on various mixed-precision strategies and their applicability to the ADER-DG component, we refer the reader to the work of Marot-Lassauzaie et al. [MB25].

4.2.1 Shock Tube

When using uniform double precision for ADER-DG and uniform single precision for the finite volume limiter, no visual difference can be seen. When changing the finite volume limiter component to half-precision, a visual difference can be noticed. While only minor differences can be noticed for float16, major differences become apparent when using bfloat16. This is likely due to wrong updates caused by rounding errors. As bfloat16 causes larger rounding errors due to its lower precision, the error grows larger over time. At the end of the simulation, float16 is still able to reach a physically acceptable state, whereas bfloat16 does not. Due to its rounding errors, bfloat16 is unable to achieve any form of equilibrium. A visualization of the final states can be seen in figure 4.5.



(a) float64

(b) float32

(c) float16

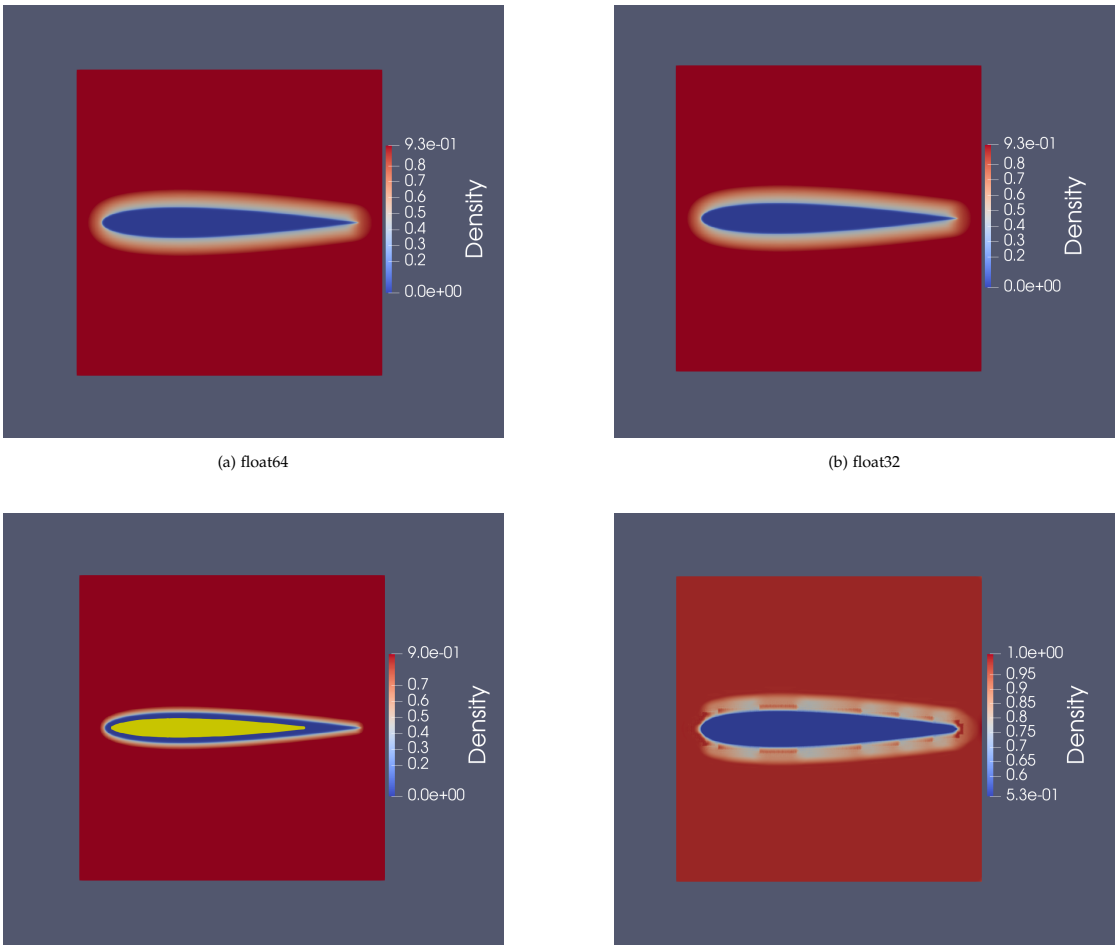
(d) bfloat16

Figure 4.5: Final states of Shock Tube Simulation (see 4.1.1) with ADER-DG solver in double precision and finite volume solver and limiting kernel in uniform precision as specified under each subfigure

4.2.2 NACA Airfoil

For the NACA Airfoil simulation, no visible difference can be seen between double and single precision. In the case of half-precision, similar behavior can be noticed at first. As expected, float16 is able to compute the solutions with a few minor differences, while larger differences become visible with bfloat16. But the implementation using float16 crashes at some point, due to NaN (not a number) values. The problem here stems from the limited range of numbers float16 can store. The computation of the flux requires the density to appear in the denominator. Therefore, low densities can cause large numbers that the float16 can not store. This results in NaN values, causing the

simulation to crash. The final states of the simulations can be seen in figure 4.6. The yellow bit in figure 4.6c corresponds to NaN values.



(a) float64

(b) float32

(c) float16

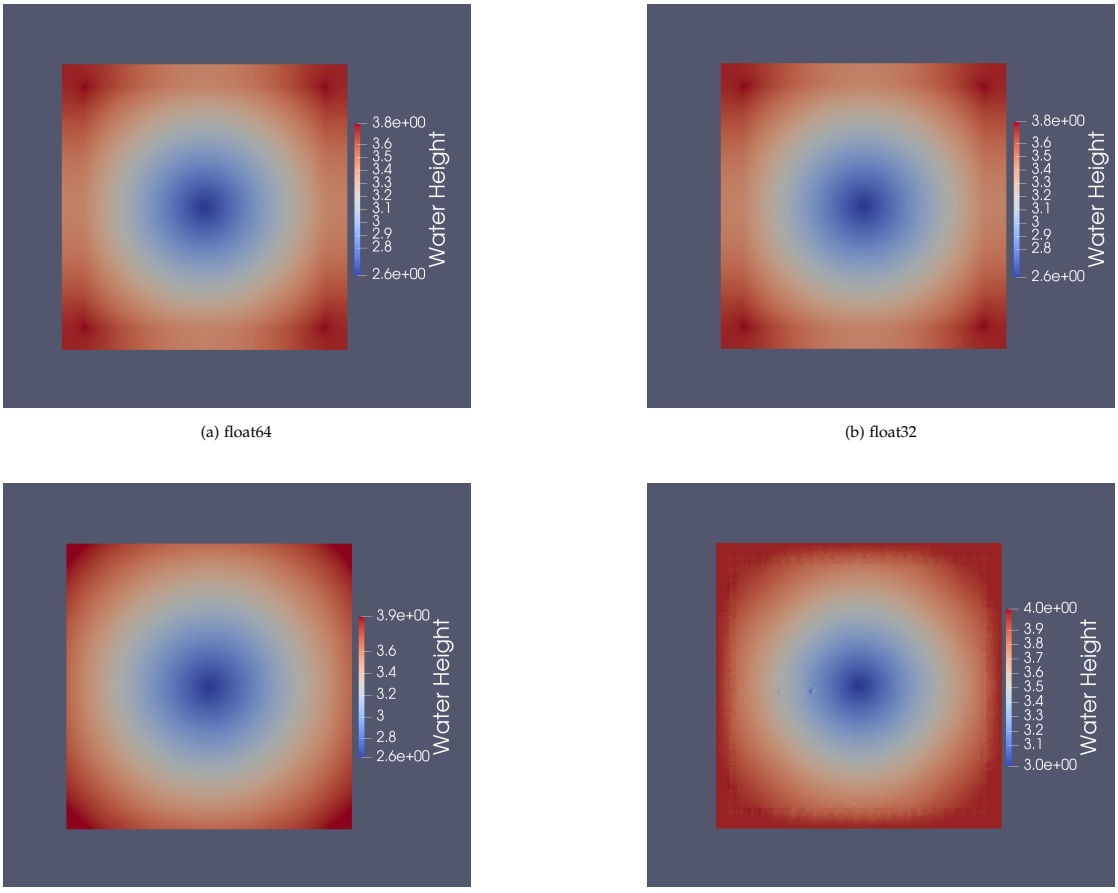
(d) bfloat16

Figure 4.6: Final states of NACA Airfoil Simulation (see 4.1.1) with ADER-DG solver in double precision and finite volume solver and limiting kernel in uniform precision as specified under each subfigure

4.2.3 Shallow Water

For this simulation, there is also no visible difference between double and single precision. Both have very similar behavior and also reach the same expected equilibrium state. The implementation using float16 also shows similar behavior at first, but fails to reach an equilibrium. Due to rounding errors, the boundary is not able to correctly implement the outflow of the waves, leading to a state where the water never reaches an equilibrium. The same is true of the implementation using bfloat16. But in contrast

to float16, the problem at the boundary caused by rounding errors can be seen almost immediately at the start of the simulation. Also, the bfloat16 implementation introduces nonphysical influences. Over time, the water height grows slightly. Because bfloat16 has larger rounding errors compared to float16, it introduces inaccuracies that significantly affect the system's physical behavior. In this specific case, bfloat16 seems to round up certain values, causing more water to accumulate over time. A visualization of the ending state of the simulation can be seen in figure 4.7.



(a) float64

(b) float32

(c) float16

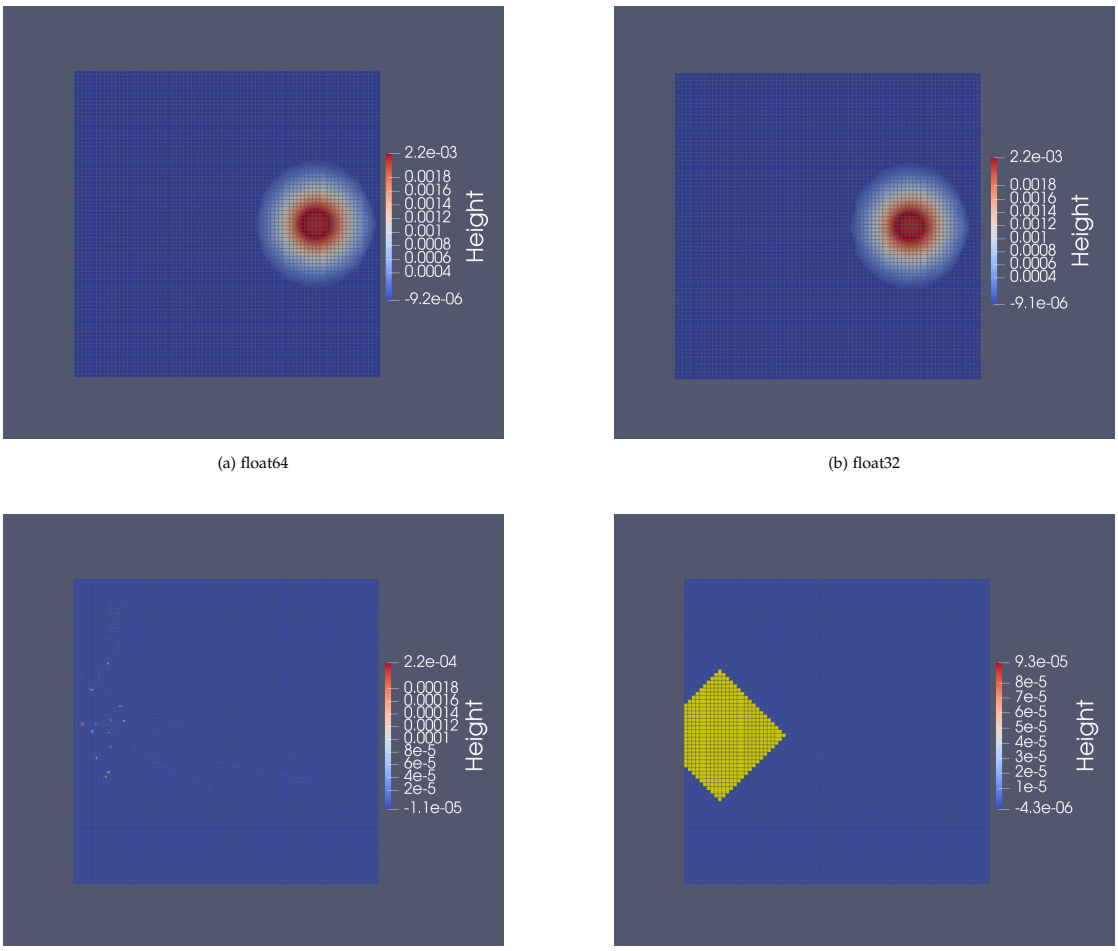
(d) bfloat16

Figure 4.7: Final states of Shallow Water Simulation (see 4.1.3) with ADER-DG solver in double precision and finite volume solver and limiting kernel in uniform precision as specified under each subfigure

4.2.4 Landslide

The Landslide simulation also shows no visible difference between double and single precision. The behavior, as well as convergence over time, is almost identical. In

contrast to the other simulations using float16, the Landslide simulation shows strong non-physical behavior. Almost instantly, at the start of the simulation, the granular material seems to disappear. This is likely caused by the fact that very small values develop in this simulation that cannot be captured by float16. The implementation using bfloat16 crashes immediately after the first step. This is probably because the solution is not computed or transferred correctly. Either zero or a negative value is used in a computation of the next step, producing NaN values. The ending states for these mixed-precision implementations are visualized in figure 4.8.



(a) float64

(b) float32

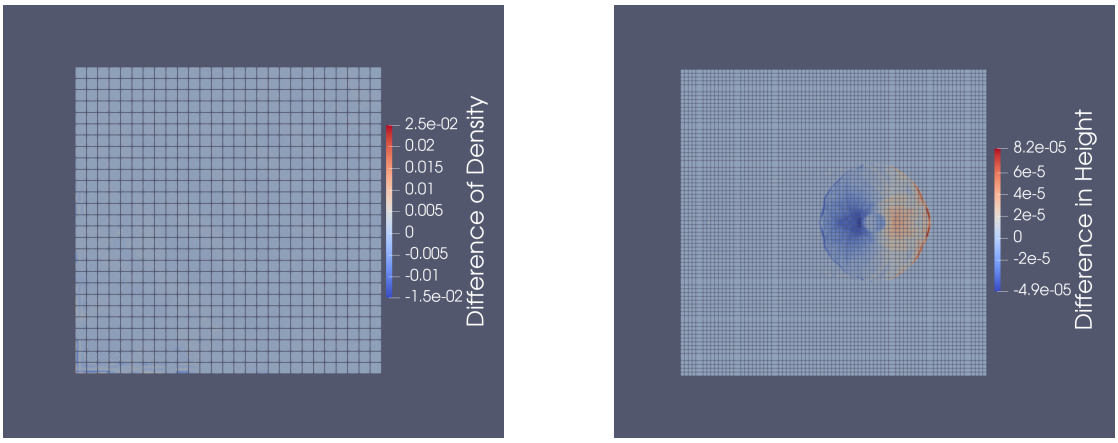
(c) float16

(d) bfloat16

Figure 4.8: Final states of Landslide Simulation (see 4.1.4) with ADER-DG solver in double precision and finite volume solver and limiting kernel in uniform precision as specified under each subfigure

4.3 Error Evaluation and Insights

The error of the mixed precision implementations using single precision in comparison to double precision stays fairly low and can be considered negligible. The behavior and convergence of the simulations stayed the same across all 4 addressed scenarios. The error margin stayed consistently in the scale of 10^{-5} with very few outliers. As a point of reference, the scope of the error of the density for the Shock Tube simulation and the Landslide simulation can be seen in figure 4.9. Notice that some outliers in the bottom left part of the Shock Tube simulation can be seen, where the error grows to the scale between 10^{-3} and 10^{-2}.

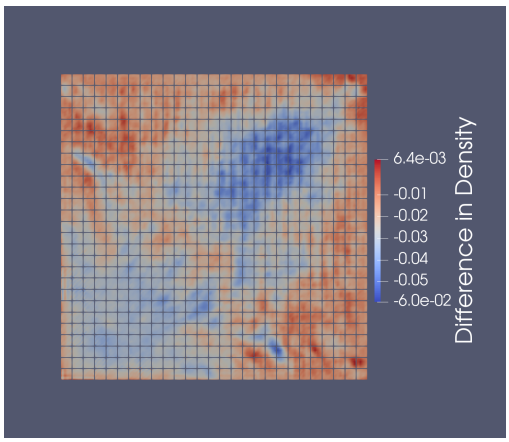


(a) Shock Tube Simulation

(b) Landslide Simulation

Figure 4.9: Plotted difference between finite volume limiter in uniform double and single precision with ADER-DG solver in double precision

All half-precision implementations, except the implementation of the Shock Tube simulation in float16, lead to unphysical properties or complete failure. Even though the Shock Tube simulation in half-precision float16 is visually close to the implementations in double and single precision, the error is still rather large. Here, the difference in density in some areas reaches a margin in the scale of 10^{-2}. A visual representation of the error is shown in figure 4.10.



(a) Shock Tube Simulation

Figure 4.10: Plotted difference between finite volume limiter in uniform double and half-precision (float16) with ADER-DG solver in double precision

The mixed-precision implementations were also tested with the ADER-DG component in single precision. For every case presented, the behavior is almost identical.

5 Conclusion and Future Work

In this work, we implemented a mixed-precision finite volume limiter for the ADER-DG algorithm. We evaluated the influence of using double, single, and half-precision on the finite volume limiter by applying the implementation on four different scenarios. The results show that using single precision has little difference in comparison to double precision. Half-precision overall is not sufficient to accurately compute physical properties modelled by the problems at hand. Both float16 and bfloat16 are also prone to causing the program to crash, due to their range or precision.

In the future, the finite volume limiter should still be tested with a bigger variety of problems and initial conditions. It could be interesting to examine the behavior when changing the polynomial order of the ADER-DG component or increasing the number of cells in the domain. Another option could be to combine the different mixed-precision implementations with different mixed-precision implementations of the ADER-DG component. Analyzing the runtime and memory usage is also left open for future work.

List of Figures

List of Tables

Bibliography

[Dum+14] M. Dumbser, O. Zanotti, R. Loubère, and S. Diot. "A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws." In: *Journal of Computational Physics* 278 (2014), pp. 47–75. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2014.08.009.

[GMM15] M. J. Grote, M. Mehlin, and T. Mitkova. "Runge–Kutta-Based Explicit Local Time-Stepping Methods for Wave Propagation." In: *SIAM Journal on Scientific Computing* 37.2 (2015), A747–A775. DOI: 10.1137/140958293. eprint: https://doi.org/10.1137/140958293.

[Gup+15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. "Deep Learning with Limited Numerical Precision." In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1737–1746.

[Int24] International Organization for Standardization. *ISO/IEC 14882:2024: Programming Languages — C++*. https://www.iso.org/standard/83626.html. C++23 standard specification. 2024.

[MB25] M. Marot-Lassauzaie and M. Bader. *Mixed-Precision in High-Order Methods: the Impact of Floating-Point Precision on the ADER-DG Algorithm*. 2025. arXiv: 2504.06889 [math.NA].

[Mic+17] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. "Mixed Precision Training." In: *CoRR* abs/1710.03740 (2017). arXiv: 1710.03740.

[Rei+20] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020), p. 107251. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2020.107251.

[Ron08] A. Ronacher. "Jinja2 documentation." In: *Welcome to Jinja2—Jinja2 Documentation (2.8-dev)* (2008).

[Shu09] C.-W. Shu. "Discontinuous Galerkin methods: general approach and stability." In: *Numerical solutions of partial differential equations* 201 (2009), pp. 149–201.

[TT02] V. A. Titarev and E. F. Toro. "ADER: Arbitrary High Order Godunov Approach." In: *Journal of Scientific Computing* 17 (2002), pp. 609–618. DOI: 10.1023/A:1015126814947.

[Wei19] T. Weinzierl. "The Peano Software—Parallel, Automaton-based, Dynamically Adaptive Grid Traversals." In: *ACM Trans. Math. Softw.* 45.2 (Apr. 2019). ISSN: 0098-3500. DOI: 10.1145/3319797.

[Zan+15] O. Zanotti, F. Fambri, M. Dumbser, and A. Hidalgo. "Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting." In: *Computers & Fluids* 118 (2015), pp. 204–224. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2015.06.020.