Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Lucas Schleuß

born in Hagen, Germany

22.10.2024

Hydrodynamic Simulations using Finite Volume and Discontinuous Galerkin Methods on Cartesian and Voronoi Meshes

This Bachelor Thesis has been carried out by Lucas Schleuß at the
Institute of Theoretical Astrophysics in Heidelberg
under the supervision of
Dr. Dylan Nelson and Dr. Chris Byrohl

Abstract

We focus on the numerical methods for solving hydrodynamic equations, including advection, the shallow water equations (SWE), and the Euler equations in one and two dimensions. In the first part, we discuss a data structure for an unstructured mesh and implement different algorithms for Voronoi mesh generation, relaxation, and the inclusion of internal boundaries. The finite volume method is introduced, covering Riemann solvers, the CFL-stability-criterion, various boundary conditions, and a second-order scheme with two different slope-limiters. Simulation results are presented to verify the correctness of the implementation on Cartesian and Voronoi meshes, including tests for advection, SWE, and Euler equations. Further, we explore classical hydrodynamic problems, such as the Kelvin-Helmholtz and Rayleigh-Taylor instabilities and qualitatively observe phenomena, such as vortex shedding and the flow around an airfoil. After that, we transition to discontinuous Galerkin methods and implement a scalar upwind advection scheme for a Cartesian mesh. We conclude with a discussion of results and future work.

Zusammenfassung

In dieser Arbeit befassen wir uns mit den numerischen Methoden, um hydrodynamische Gleichungen, wie die Advektionsgleichung, die Flachwassergleichungen (SWE) und die Euler Gleichungen in einer und zwei Dimensionen zu lösen. Im ersten Teil befassen wir uns mit einer Datenstruktur für ein unstrukturiertes Gitter und implementieren verschiedene Algorithmen um Voronoi Gitter zu generieren, sie gleichmäßiger zu machen und Objekte in das Gitter einzufügen. Wir führen die Finite Volumen Methode ein, behandeln dabei Riemann-Löser, das CFL-Stabilitätskriterium, verschiedene Randbedingungen und ein Schema zweiter Ordnung mit zwei verschiedenen Steigungslimitierern. Um die Richtigkeit der Implementierung auf kartesischen und Voronoi Gittern zu gewährleisten, werden Simulationsergebnisse präsentiert, die die Advektsionsgleichung, die Flachwassergleichungen und die Euler Gleichung testen. Desweiteren betrachten wir klassische hydrodynamische Probleme, wie die Kelvin-Helmholtz und die Rayleigh-Taylor Instabilitäten und beobachten qualitativ Phänomene wie die Wirbelablösung an einen Zylinder und die Strömung um ein Tragflächenprofil. Danach wechseln wir zu der diskontinuierlichen Galerkin Methode und implementieren einen Algorithmus, der die skalare Advektionsgleichung mit einem Upwind-Schema auf einem kartesischen Gitter löst. Zum Schluss werden die Ergebnisse noch einmal zusammen gefasst und enden wir mit einem Ausblick auf zukünftige Arbeiten.

Contents

1 Introduction

1.1 Simulations in astrophysics

Hydrodynamic simulations are a crucial tool in astrophysics. Astonishingly, on large scales, the gas dynamics in space and even stellar dynamics of galaxies follow the same fundamental rules as the river nearby. The large scale of astrophysical objects and the immense time scales (often millions of years) make it practically impossible to design experiments with galaxies or to observe events like mergers from start to finish. Therefore, simulations are the key to understanding the formation, evolution and dynamics of various astrophysical objects. Simulations are widely used, from understanding the structure of protoplanetary discs to studying galaxy formation and the structure of the cosmic web. Most modern codes solving these equations are massively parallelized and run on supercomputers. Even with such high performance, the simulations can take months to complete. This creates a huge incentive to accurately and efficiently solve the underlying equations. (Springel, 2010)
In this thesis we want to dive deep into some of the underlying methods that allow us to do such simulations. We will mainly focus on the theory and implementation (in C++) of finite volume (FV) methods and also take a look at discontinuous Galerkin (DG) methods. Finite volume methods divide the space into cells, each storing the cell average of the quantities we are simulating (e.g. density, velocity, ...). The equations are then solved by computing fluxes, based on which the cell averages are updated. Many finite volume codes are written with second-order accuracy. This means doubling the resolution decreases the error by a factor of four. The improvement in accuracy using second-order is huge and the implementation requires only to know the cell neighbours. Even higher order accuracy with this method, however, requires to know many neighbouring cells (e.g. neighbours of neighbours) and becomes complicated quickly. (Springel et al., 2020)
The AREPO code is a prominent example of the use of finite volume methods in astrophysics (Springel, 2010). It is used by the IllustrisTNG collaboration to simulate the gas dynamics part of cosmological galaxy formation (see Fig. 1). In the finite volume section, we will follow some of the ideas of that code.

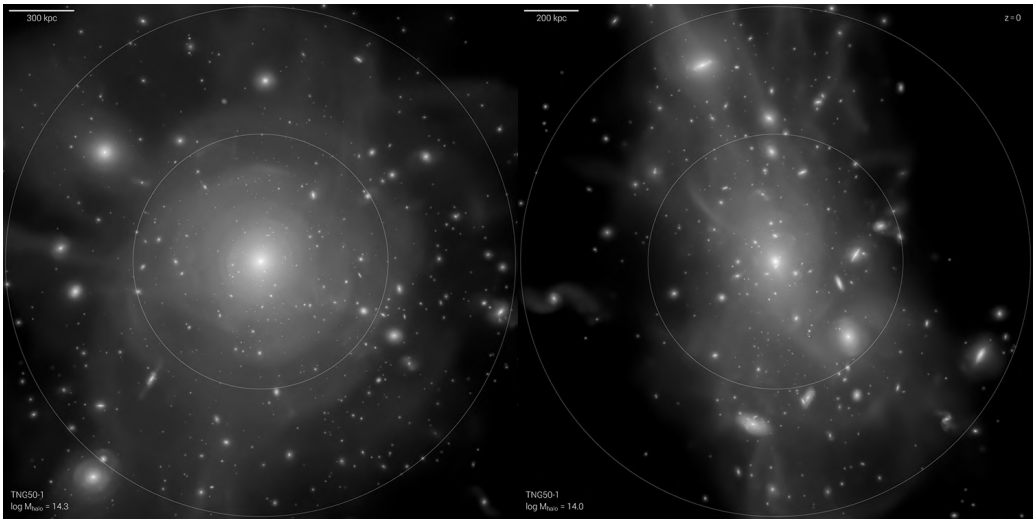

Figure 1: Two large galaxy clusters out of the TNG50 Simulation, that uses the finite volume code AREPO. Credit: TNG Collaboration, (Joshi et al., 2020)

Discontinuous Galerkin methods instead are still less common, but have been gaining popularity in recent years. This is due to their systematic way of achieving arbitrary higher-order convergence while keeping the stencil small. In fact, one only needs to know the immediate neighbors. DG is a finite element method, where cells are called elements and instead of evolving element averages, functions representing the quantities are evolved. Contrary to other finite element methods like continuous Galerkin, the functions in DG are allowed to be discontinuous at element boundaries (Knezevic, 2012). We will follow some ideas of the astrophysical DG code called TENET later on (Schaal, 2016).
Even though the two methods are quite different, both methods are grid-based schemes. We will look at Cartesian and Voronoi meshes. A Voronoi mesh is a grid where the space is divided into cells based on the distance to a set of seedpoints. Each cell contains the region closer to its seedpoint than any other seedpoint. Voronoi, along with other unstructured meshes (e.g. triangular), has the advantage of being highly flexible. For example, the resolution can be freely adapted without requiring additional mesh refinement. Implementing custom inner boundaries, like an airfoil, also works naturally with unstructured meshes. One particular reason Voronoi cells have gained attention in astrophysics is the ability to let the mesh move with the flow. This leads to a Galilean invariant simulation while maintaining advantages like better shock resolution compared to other gridless methods (e.g., Smoothed Particle Hydrodynamics) (Springel, 2010). That is especially useful for large-scale galaxy formation simulations, where no global reference frame can be defined, in which all galaxies are stationary. The lack of Galilean invariance reduces the simulation accuracy for galaxies moving fast relative to the arbitrarily fixed mesh. In addition to the Galilean invariance, the flowing cells also increase resolution in regions where the density is increasing and reduce resolution in areas where the density is decreasing. This saves unnecessary computations in low-density regions while improving the resolution in dense areas. However, the computational cost to regenerate the Voronoi mesh after every timestep is also significant (Springel, 2010). For simplicity, we will not focus on the moving mesh part and will stay in 2D, mainly due to computational power limitations.

1.2 Project overview

The main hydrodynamic equations we will try to model are advection, the shallow water equations (SWE) and the Euler equations. After that we will look at the implemented data structure of the mesh and focus on how to fast and reliably generate Voronoi meshes. We then look at methods to relax the mesh and add internal boundaries to represent objects in the domain. After that, we will introduce finite volume methods, discuss Riemann solvers, stability, boundary conditions and the theory behind implementing a second-order method. We will then look at the implementation results on Cartesian and Voronoi meshes, starting with advection, followed by shallow water equations and the dam break problem. In the next part, we use the analytically known shock tube test to verify our solutions to the Euler equation. We look at some classical test problems like the Kelvin-Helmholtz instability, the Rayleigh-Taylor instability, some 2D Riemann problems and more. Next, we leave finite volume methods and start with the theory of discontinuous Galerkin methods. After looking at a 1D implementation of advection we generalize the method to 2D DG advection on a Cartesian mesh. Finally, we will conclude with a summary of the results and an outlook on ideas for further work. Even though this project focuses only on widely known algorithms, there are many areas which are still

actively researched. For example, an efficient way to do Discontinuous Galerkin methods on a moving Voronoi mesh is still not entirely solved (Mocz et al., 2013), (Gaburro et al., 2020), (Walter Boscheri et al., 2022).
All the code written for this project and also some animations of the plots, shown later on, are available on github.com/lucas56098/hydro_bsc_project.

1.3 Hyperbolic equations

The equations that we will try to model are the advection equation, the shallow water equations and the Euler equations. These all have in common that they are hyperbolic partial differential equations (PDE), each conserving some quantities (e.g. energy and momentum). But why are they called hyperbolic? Partial differential equations are divided into three main classes: hyperbolic, parabolic and elliptic (Knezevic, 2012).

equation type	example	equation
hyperbolic	wave equation	$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0$
parabolic	heat equation	$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = c$
elliptic	Poisson equation	$\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial x^2} = c$

Table 1: Classes of hyperbolic equations and example equations.

The equations in Tab. 1 look roughly similar, but their characteristics are fundamentally different. In fact, one needs different methods to solve them. Equations of the same type (e.g. hyperbolic), however, can be solved with similar methods (Springel et al., 2020).
The names of those categories result from comparing them with a quadratic function (Knezevic, 2012). A general second-order PDE for a function $u(x,t)$ has the form

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial t} + C\frac{\partial^2 u}{\partial t^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial t} + Fu + G = 0. \tag{1.1}$$

When comparing this to a general quadratic function

$$q(x,t) = Ax^2 + Bxt + Ct^2 + Dx + Et, \tag{1.2}$$

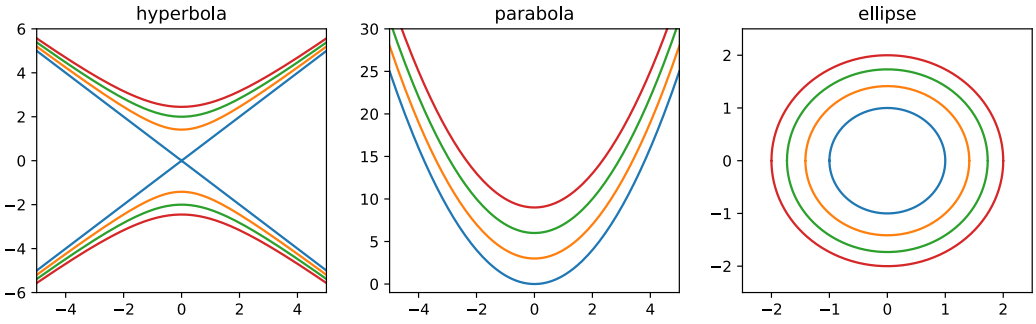

Figure 2: Corresponding quadratic functions for different PDE types.

we see that for the example differential equations one gets a hyperbola $t^2 - x^2 = c$ for the wave equation, a parabola $t - x^2 = c$ for the heat equation and an ellipse $t^2 + x^2 = c$ for the Poisson equation[1]. (Knezevic, 2012)

The wave equation can be written in terms of coupled first-order equations.

$$\frac{\partial u_1}{\partial t} + \frac{\partial u_2}{\partial x} = 0 \tag{1.3}$$

$$\frac{\partial u_2}{\partial t} + \frac{\partial u_1}{\partial x} = 0 \tag{1.4}$$

Generally, conservation laws are written in terms of first-order derivatives (Knezevic, 2012).

Advection

The easiest case of this is the advection equation. It advects an initial condition $u(\vec{x}, 0)$ with velocity $\vec{v}$ without changing it in any way. Hence the analytical solution to the advection equation is $u(\vec{x}, t) = u(\vec{x} - \vec{v}t, 0)$. The advection equation for 2D can be written as:

$$\boxed{\frac{\partial u}{\partial t} + \vec{v} \cdot \vec{\nabla} u = 0} \tag{1.5}$$

Defining the flux $\vec{F} = \vec{v}u$ we get:

$$\frac{\partial u}{\partial t} + \vec{\nabla} \cdot \vec{F} = 0 \tag{1.6}$$

More complex conservation equations like shallow water or Euler equations can be brought into a similar form, defining a state vector $\mathbf{U}$ (Springel et al., 2020). Notation-wise vectors written like $\vec{a}$ will have spatial dimension (x, y) while $\mathbf{a}$ will be a vector with the same dimension as the state vector. We then can also define a Flux vector $\vec{\mathbf{F}}$ consisting of $(\mathbf{F_x}, \mathbf{F_y})$. Replacing u and $\vec{F}$ with the corresponding state vectors one gets:

$$\frac{\partial \mathbf{U}}{\partial t} + \vec{\nabla} \cdot \vec{\mathbf{F}} = 0 \tag{1.7}$$

One interesting thing to note is that at least for the linearized case, we can rewrite the flux using the flux jacobian and the state vector: $\vec{\mathbf{F}} = \frac{\partial \vec{\mathbf{F}}}{\partial \mathbf{U}} \mathbf{U}$. Assuming we can diagonalize the flux Jacobian with real eigenvalues, we can separate any linear conservation equation written in state vector notation into multiple advection equations with constant speed (Knezevic, 2012).

$$\frac{\partial \tilde{\mathbf{U}}}{\partial t} + \vec{\mathbf{\Lambda}} \cdot \vec{\nabla} \tilde{\mathbf{U}} = 0 \tag{1.8}$$

Here $\tilde{\mathbf{U}}$ is the state vector in the eigenbasis and $\vec{\mathbf{\Lambda}} = (\mathbf{\Lambda_x}, \mathbf{L_y})$ are the diagonal flux Jacobian matricies. Those eigenvalues define the characteristics of the system and give valuable information on how the information (e.g. a small perturbation) is propagating

[1] Formally determining the category is a bit more involved and requires to look at the discriminant.

through the system. This is intuitive for a single advection equation where the characteristic is defined by the velocity $\vec{v}$. Hence, the information also travels with the velocity (Springel et al., 2020).

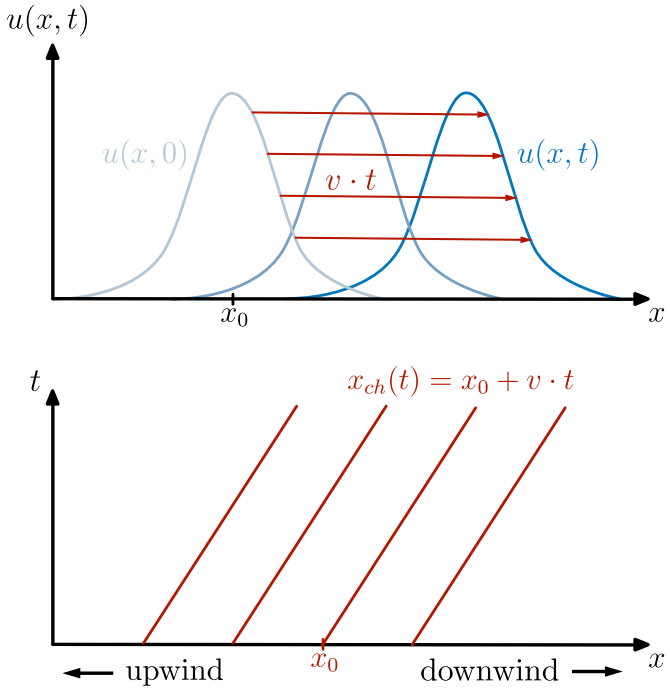

Figure 3: Top: An initial condition being advected. Bottom: The information flow is described by a characteristic. (Springel et al., 2020)

Any point will only get information from the upwind side and no information from the downwind side (see Fig. 3). It, therefore, makes sense only to use the upwind side when calculating fluxes through edges (Springel et al., 2020). In a more abstract way, this idea also works for more complicated equations, which will be discussed later in the chapter on Riemann solvers.

Shallow water equations

With this general notation, we can introduce the shallow water equations next. The shallow water equations (SWE) are a set of hyperbolic PDE that describe fluid flow under the assumption that the horizontal length scale is much larger than the vertical length scale (Dawson and Mirabito, 2008). For 2D SWE, they can be written in state vector notation as

$$\frac{\partial}{\partial t}\begin{pmatrix} h \\ hu \\ hv \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix} = 0. \tag{1.9}$$

Here h is the water column height, u is the velocity in the x-direction, v is the velocity in the y-direction and g is the gravitational acceleration (Knezevic, 2012). For large scales, this assumption holds true for the earth's ocean and the atmosphere, which is why people use it for tsunami modeling or large-scale atmospheric wind simulations (Dawson

and Mirabito, 2008). Also, flooding prediction and river hydraulics are done with SWE (Hydrotec)(Dawson and Mirabito, 2008). Additional source terms, such as an uneven ocean surface in principle, can be added to the right-hand side of the equations. The main reason why we are looking at it is as a stepping stone towards Euler equations.

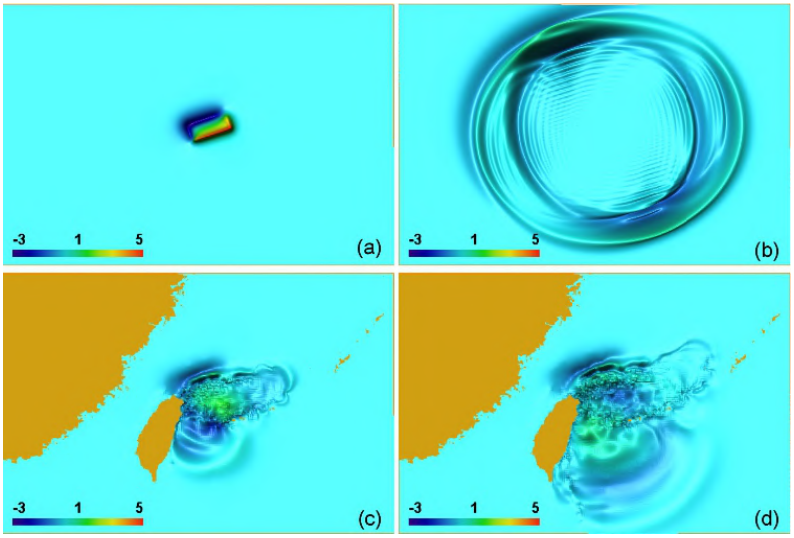

Figure 4: Tsunami modeling being done with shallow water equations. Credit: (Zhang et al., 2007)

Euler Equations

The third equations that we are going to model are the Euler equations. They are the fundamental equations of ideal hydrodynamics and describe inviscid fluids. Essentially, they are mass, momentum and energy conservation equations modeled by hyperbolic PDE and hold true for any scenario where the following assumptions are given (Bartelmann, 2021):

Scale hierarchy: The minimal fluid volume considered must contain a sufficient number of particles on the microscopic scale, that it makes sense to define macroscopic bulk properties like density, pressure, temperature and velocity. This requires that microscopic fluctuations are reliably averaged out. Further, the systems must be so much larger than that minimal fluid volume that it makes sense to describe those bulk properties using continuous fields. This is true if the free mean path of the particles λ is much smaller than the length of the minimum fluid volume l, which itself is way smaller than the characteristic length L of the system (Bartelmann, 2021).

$$\lambda << l << L \tag{1.10}$$

If this is not the case, one needs to consider gas kinematics instead of fluid mechanics.

Collisional invariance: The collisions in the system must be so frequent that, when averaged over the momentum phase space, their effects cancel out. This is true either for a huge number of collisions or no collisions at all (Bartelmann, 2021).

Viscosity and heat conduction: Additionally, Euler equations assume no heat conduction (adiabatic movement) and no viscosity. The modeled fluid, hence, needs to have

neither of these in good approximation. If this is not the case, one can generalize to the Navier-Stokes equations, which include viscosity and heat conduction (Bartelmann, 2021).

Any system that fulfills these approximations can be modeled by using the following equations (Knezevic, 2012):

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}+\frac{\partial}{\partial x}\begin{pmatrix}\rho u\\ \rho u^2+P\\ \rho uv\\ (E+P)u\end{pmatrix}+\frac{\partial}{\partial y}\begin{pmatrix}\rho v\\ \rho uv\\ \rho v^2+P\\ (E+P)v\end{pmatrix}=0 \tag{1.11}$$

These are written in state vector notation, u and v are the x- and y-velocities again and ρ is the density. E is the total energy per unit volume and P the pressure. The first equation conserves mass, the second and third momentum and the fourth conserves energy. A detailed derivation of those equations, starting from the Liouville equation, can be found in (Bartelmann, 2021). In addition to those equations, we need to choose an equation of state, giving a relation between the pressure and the state variables. For an ideal gas, this is given by

$$P=(\gamma-1)\cdot\Big(E-\Big(\frac{\rho}{2}\cdot(u^2+v^2)\Big)\Big), \tag{1.12}$$

where $\gamma=\frac{f+2}{f}$ is the adiabatic index (Knezevic, 2012).

Given those general assumptions, the Euler equations have many different applications. From aerodynamics and weather predictions to jet flows and astrophysics, these equations play a crucial role. In astrophysics, the various types of gas (e.g. intergalactic medium) fulfill the scale hierarchy and collisional invariance. The low viscosity is fulfilled in most cases and heat conduction can be added using radiative transfer (Bartelmann, 2021). This is why one can, for example, simulate gas flows in and out of galaxies, as shown in Fig. 5. Often, magnetic fields are present in astrophysics, which then generalizes the equations to the equations of magnetohydrodynamics (MHD). Interestingly, stellar dynamics can also be modeled using fluids. The scale hierarchy is given since a galaxy has an enormous amount of stars. Stars in a galaxy are, to good approximation collisionless, which is the reason why the collision integrals vanish. One then gets equations very similar to the Euler equations called the Jeans equations (Bartelmann, 2021). First, we will model the advection equation and then go on to more complex equations.

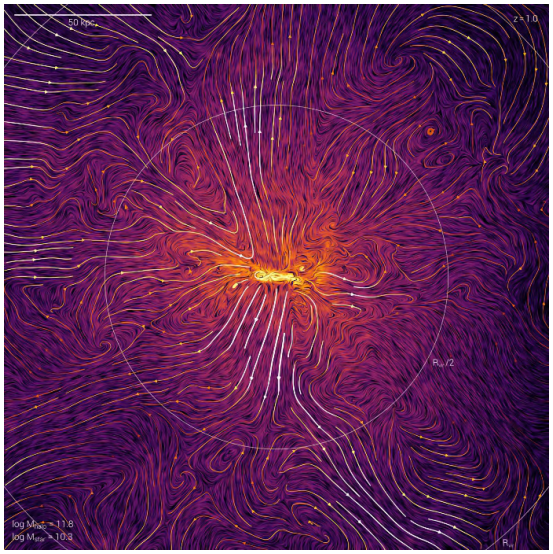

Figure 5: Halo-scale gas flows visualized around a single TNG50 galaxy. The dynamics of this system can be modeled in part by Euler equations. Credit: TNG Collaboration, (Nelson et al., 2019)

2 The Mesh

2.1 Data structure

The mesh stores the entire simulation data for the current timestep. We try to keep the mesh very general to use the same mesh class for all the different equations. The mesh consists of cells, as shown in Fig. 6. Cells have a seedpoint, an ordered list of edges and store the problem-specific cell quantities. Ordering here and anywhere else in the code is always clockwise. This is needed to be consistent with the definition of normal vectors (Springel, 2010). The cell edges include the start and endpoint as well as the length. In addition to that, it stores if it has a neighbor and optional a pointer to that neighbor cell.

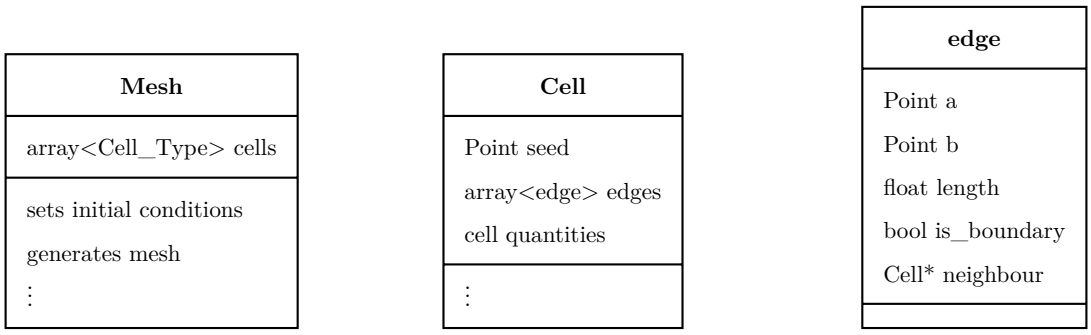

Figure 6: General structure of the mesh. Specific functions and helper functions are not included here.

C++ templating is used to allow changing the cell type at compile time depending on the equations (e.g. `Q_Cell`, `SWE_Cell`, `Euler_Cell`, `DG_Q_Cell`). For example, a `Q_Cell` just stores a single scalar quantity Q as cell quantity and can be used for the advection equation. An `Euler_Cell` instead stores density, velocity and energy as well as the adiabatic index. The mesh also has functions to set the cell quantities to different initial conditions and to generate the geometric mesh structure.

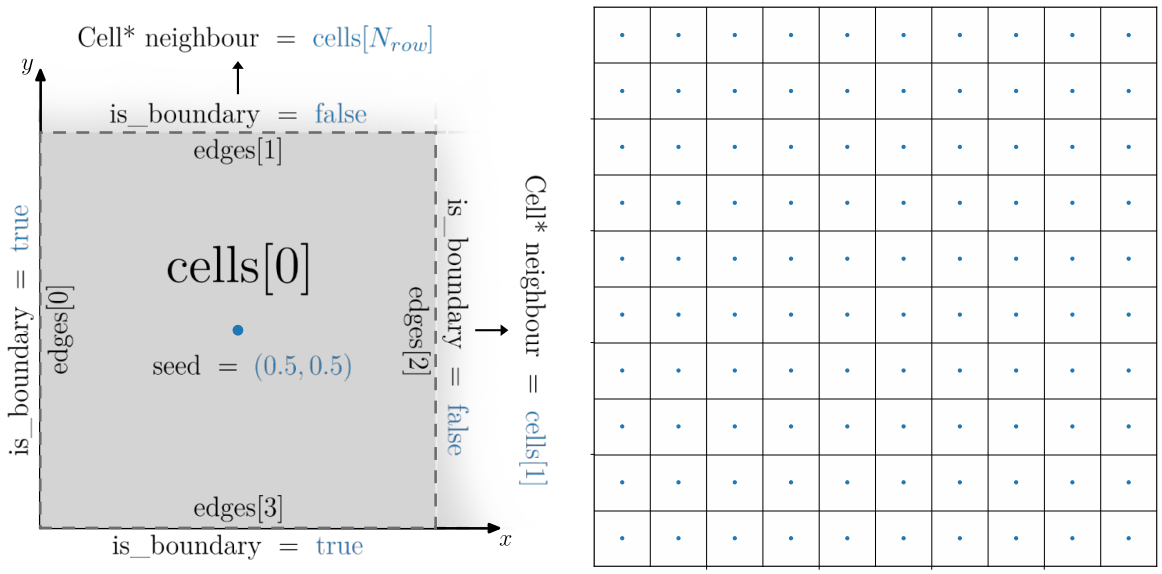

Figure 7: Left: Data stored in a cell. Right: Cartesian mesh

Mesh generation for a Cartesian and Voronoi mesh are implemented. However, the structure could be used for arbitrary geometries. In Fig. 7, a Cartesian mesh is shown, consisting of cells with evenly spaced seedpoints and four faces each. For edges touching the boundary, one sets the `is_boundary` flag to `true`, and for all other edges, sets the neighbor relations. All grids we are going to generate and use in simulations will be on the $(0, 0)$ to $(1, 1)$ square. This is because double precision is highest in that area (Springel et al., 2020). In theory, one could always scale back the mesh after simulating. Since we will not actively use units, we will skip the rescaling part. All plots will, therefore, be of that square or subregions in that square. Ticks on plots will generally be 0.2 arbitrary units ($[a.u.]$) apart.

2.2 Generating a Voronoi mesh

Next, we will focus on how to generate a Voronoi mesh with a given set of seedpoints. A Voronoi cell is a polygonal region surrounding a specific seedpoint in space, containing all locations closer to that point than any other seedpoint. The edges of a Voronoi cell are a part of the perpendicular bisectors to the neighboring points (Springel, 2010). We will call the bisectors halfplanes in the following. In theory, any pair of seedpoints then defines a halfplane that we will later use to construct the Voronoi cells. A Voronoi mesh, as in Fig. 8, is the combination of all Voronoi cells.

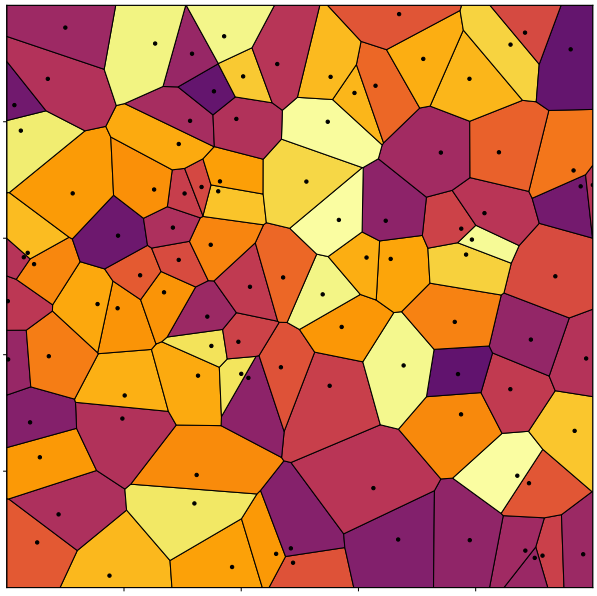

Figure 8: A Voronoi mesh. The cells encompass all regions closest to their seeds.

Generating a Voronoi mesh fast and reliable is not trivial. Mesh generation, especially for moving mesh codes, can take a significant amount of time compared to the simulation (Springel, 2010). We, therefore, want to implement an algorithm that scales optimally. There are two existing types of algorithms: direct ones and indirect ones. Direct algorithms generate a Voronoi mesh directly from the seedpoints. Indirect algorithms use the fact that a Delunay triangulation is the geometric dual of a Voronoi mesh and employ algorithms to get the Delunay triangulation before converting it to Voronoi (Springel, 2010). We will focus on direct algorithms. For that, we will start with a conceptually easier naive halfplane intersection algorithm that scales with $\mathcal{O}(N^2)$ (Sacristan, 2019). After

that, we introduce a more advanced point insertion algorithm which has an improved scaling of $\mathcal{O}(N \log N)$ (Sacristan, 2019), (Springel, 2010).
Before introducing the algorithms, some essential operations should be discussed in advance. First, there will be many situations where two halfplanes will intersect. In 2D, this can be done easily by solving a linear equation system and evaluating some determinants. For the intersection, we store the intersecting point and the intersecting halfplane. We also store the signed distance of the intersection to the midpoint of the current halfplane. The positive distance here means in a clockwise direction. Using that information, we can intersect many halfplanes and find the intersection with the smallest positive signed distance relative to either the midpoint or the last vertex, depending on the situation.

Naive halfplane intersection

To generate a Voronoi cell with naive halfplane intersection (see. Fig. 9), the algorithm starts with the halfplane closest to the seed, which will be the first edge. Starting from there, one finds the halfplane intersection with the smallest positive distance relative to the midpoint of that first edge. This is done by intersecting the first edge with all other existing halfplanes relative to the seed. The intersecting halfplane will be the next edge that can be stored and their intersection will be a vertex. To repeat the process, the next edge becomes the current edge and instead of the midpoint, we now use the vertex for the signed distances. Then the smallest positive intersection relative to the vertex is calculated and we continue as above. This process repeats until one returns to the first halfplane. (Sacristan, 2019)

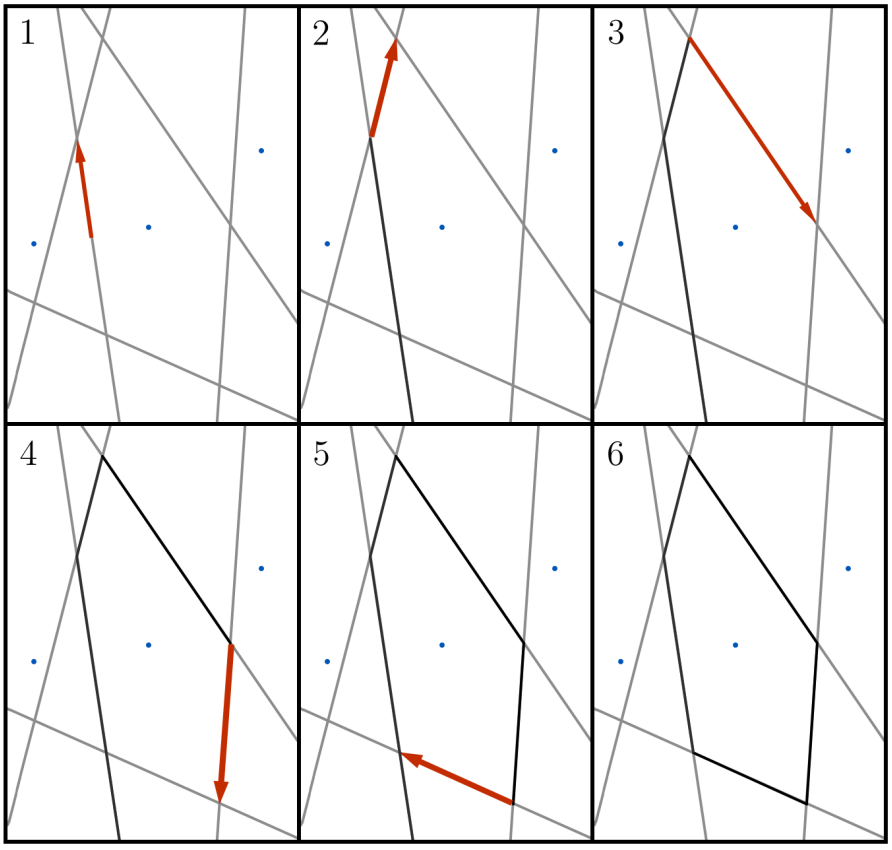

Figure 9: Naive halfplane intersection step by step

Boundary handling here is reached by adding four halfplane boundaries to the total list of halfplanes to be checked. This algorithm, in total, needs $\mathcal{O}(N^2)$ steps. We will check this later in the performance benchmark. Interestingly, this algorithm generates a cell once and then leaves it that way for the entire algorithm (see Fig. 10), which is conceptually different to the point insertion algorithm.

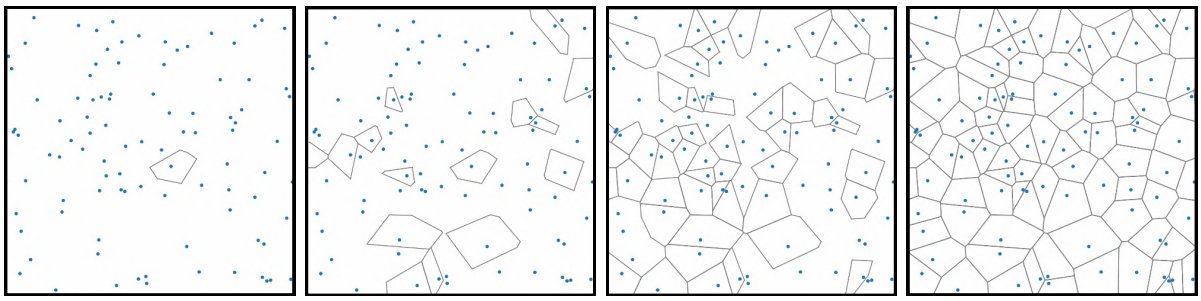

Figure 10: Mesh generation with naive halfplane intersection. From left to right, more and more cells are generated. All seedpoints are there from the start.

Point insertion

The point insertion algorithm is conceptually more difficult and requires a focus on adapting the neighboring cells and respecting boundary conditions. The algorithm inserts one seedpoint after another and then adapts the mesh to include that new seedpoint (Sacristan, 2019). Conceptually, it is sufficient to look at inserting one seedpoint because this process is repeated many times. If we want to insert one seedpoint into an existing mesh, we first need to find out which cell we are currently in. For that, we use a `find_cell()` function that starts from a specified cell and from there jumps step by step towards the new seedpoint until it can't get closer. The cell it is in then must be the cell where the seedpoint is in as well. This process can be massively optimized later on (see Sect. 2.2). When we know the cell we are in, we can construct the halfplane between the new seedpoint and the seedpoint of the cell we are in. This halfplane will be an edge of the new cell. Now, we look at the edges of the cell we are in and intersect them with that halfplane. We then want to find the intersection with the smallest positive distance relative to the midpoint. With that intersecting edge, we know which cell will be the next to use for the construction. Making a halfplane between the next cell's seedpoint and the new cell's seedpoint gives us the next edge for the new cell. Instead of using the midpoint, we again use the vertex of the last intersection for the relative distances. This process can be repeated until we reach the first cell we started in again. Now that the new cell is constructed, we must adapt the surrounding cells by clipping their edges according to where the intersections have been. The whole process is visualized in Fig. 11.

Boundary handling here is more difficult than in the first algorithm because there is no cell we can enter when reaching a boundary. This is solved by staying on the boundary as long as a leaving condition is not satisfied and afterwards continuing with the algorithm.

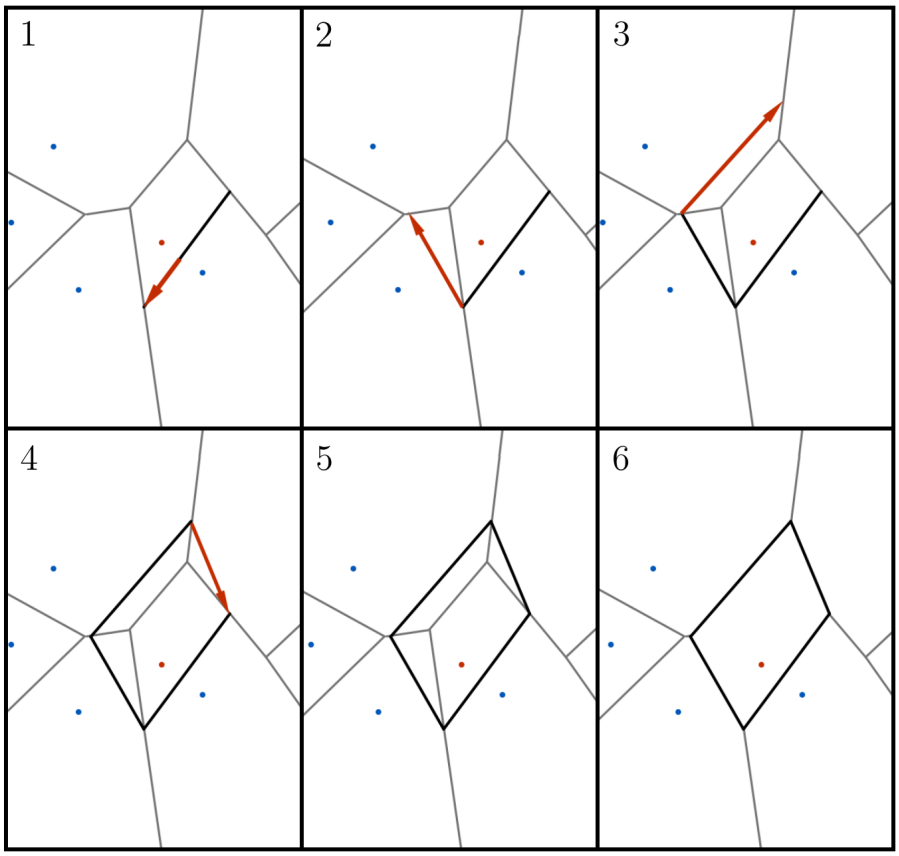

Figure 11: Point insertion algorithm step by step.

The leaving condition is the halfplane between the new seedpoint and the seedpoint of the cell we are checking, intersecting with the boundary inside of the checked cell (see Fig. 12).

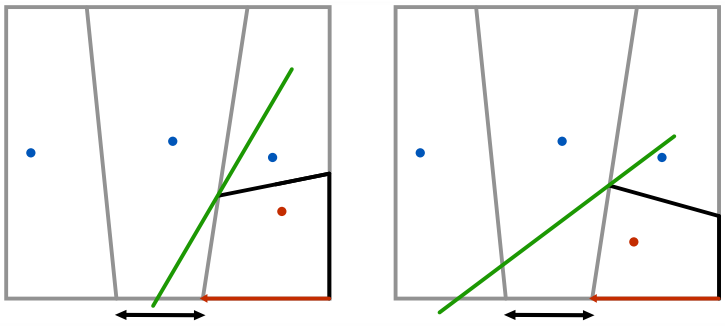

Figure 12: Left: leaving condition is fulfilled. Right: leaving condition is not fulfilled.

Given the optimized cell finding, the point insertion algorithm scales with $\mathcal{O}(N \log N)$ (Springel, 2010). In contrast to the naive algorithm, it generates the cells only based on the already inserted cells, which means that cells will change multiple times after being inserted (see Fig. 13).

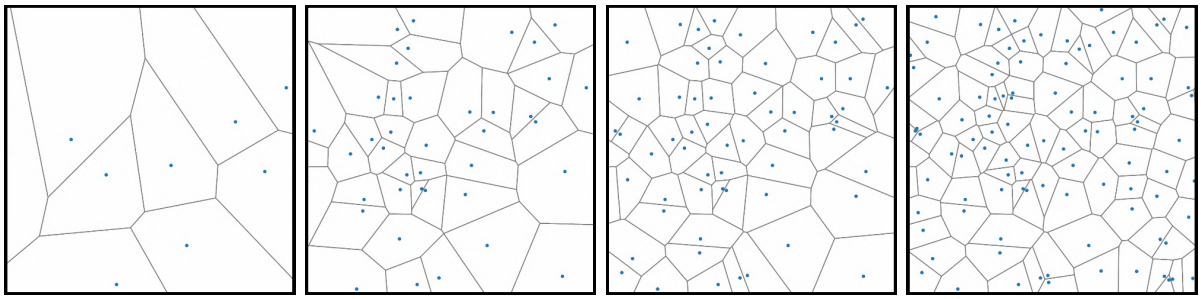

Figure 13: Mesh generation with random point insertion. From left to right, more and more cells are inserted into the mesh.

Presorting Seedpoints

As mentioned before, we need to massively optimize the speed of the `find_cell()` function to achieve $\mathcal{O}(N \log N)$ scaling (Springel, 2010). For an unsorted pointset, this function needs to do many steps to reach the cell in which the new seed is located. To reduce this, we sort the seedpoints so that points that are close together are also close together index-wise. This can be done in many different ways, for example, using space-filling curves (e.g., Peano Hilbert (Springel, 2010)). We will instead use a grid-based sorting illustrated in Fig. 14. Within each grid cell, the seeds will stay unsorted, while the seeds will be sorted according to which grid cell they are in. The grid resolution can be changed as needed. This is relatively easy to implement and reduces the number of steps to the next cell to very few. A sorted mesh generation with point insertion looks like Fig. 15.

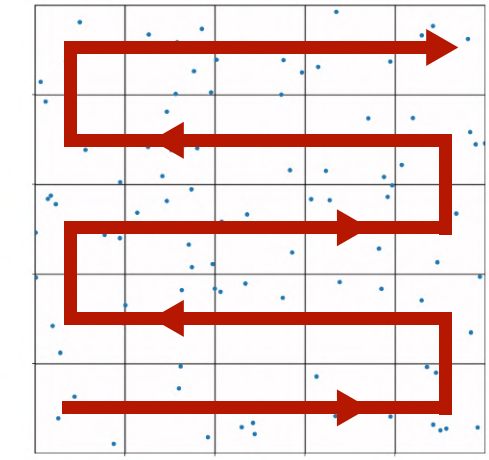

Figure 14: Grid-based sorting. The points stay unsorted inside grid cells, while the Grid is sorted along the red arrow.

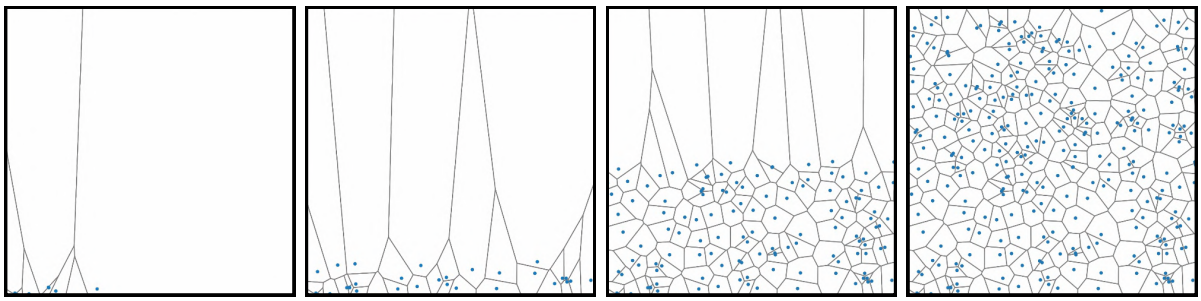

Figure 15: Mesh generation with presorted point insertion. From left to right, more and more cells are inserted into the mesh.

Degeneracies

Another important thing to consider when reliably generating a mesh are degeneracies (Springel, 2010). A degeneracy occurs when a vertex has more than three equidistant nearest seeds. Thus, the function to find the smallest positive intersection finds two or more halfplanes with the exact same smallest distance. For a random point set, degeneracies are extremely rare, but they are important to get right for special cases. The handling of degeneracies is very limited in the implemented code. The naive halfplane intersection algorithm includes handling degenerate cases by comparing the direction of the degenerate halfplanes and choosing the right one. However, artifacts occur in highly degenerate cases, like larger uniform grids. This seems strange, and thus, this degeneracy handling must be handled with care. The point insertion algorithm at the moment does not come with degeneracy handling. This, however, is not problematic for us because we mainly use random seedpoints, for which practically no degeneracies occur. Even in cases where we implement highly degenerate grids, we can always add a small random perturbation to the seeds to avoid any problems. For almost degenerate cases up to relative distances of 10^{-13}, the code runs without problems. Fig. 16 is an example of an almost uniform grid. The Cartesian mesh we use in the simulations is generated with a separate function, so there is no need for such almost uniform grids.

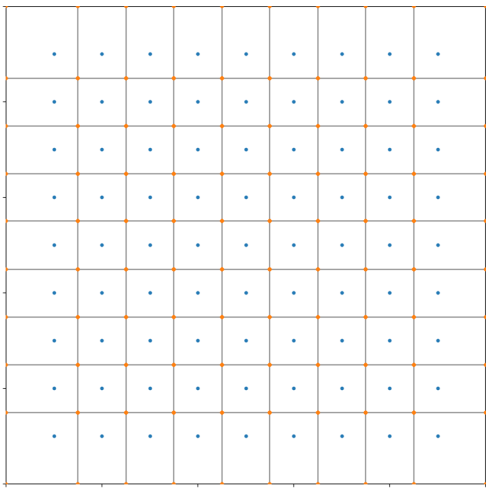

Figure 16: Almost uniform grid generated using point insertion. The seedpoints vary by around 10^{-13} from uniform.

Correctness checks

We want to verify that our mesh at least satisfies some of the properties we expect from the Voronoi mesh. This is important to verify that the algorithms work as expected. We check the following properties:

- Equidistance: Every vertex should have at least three equidistant seedpoints. (Definition of a vertex)
- Area: The total area of the mesh should equal the box size up to double precision.
- Neighbours: Every neighbor of a cell should have the cell as a neighbor as well.

Performance and memory usage

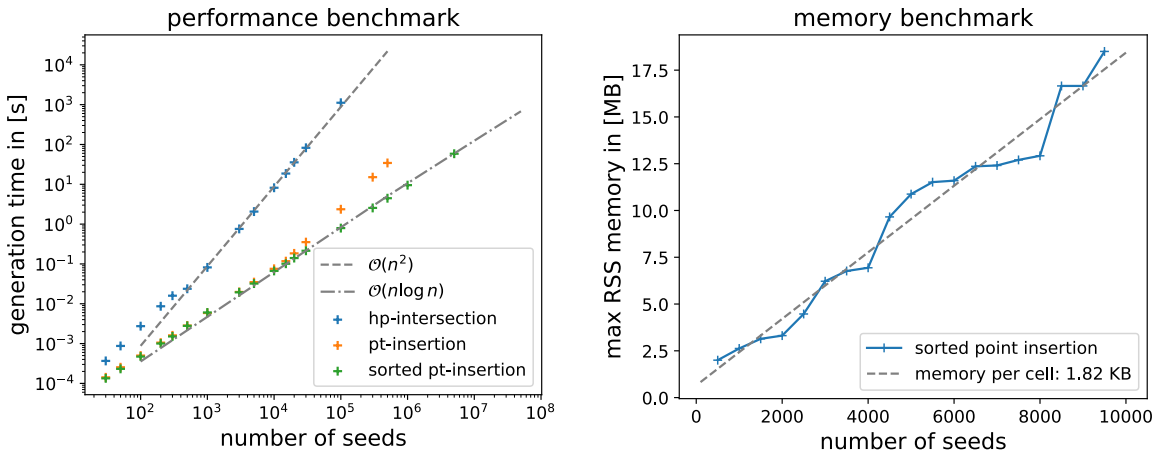

Figure 17: Left: performance of naive halfplane versus point insertion, Right: memory benchmark

In Fig. 17, the generation time on a MacBook Pro M1 was plotted as a function of seedpoints to generate. As one can see the algorithms scale as expected. The sorting of the seedpoints, according to the grid-based sort, is the final piece in the puzzle to achieve $\mathcal{O}(N \log N)$ scaling. Otherwise, for large seedpoint sets, the cell finding function scales worse and takes many steps to reach the cell where the seedpoint is in. The memory grows approximately linear, which is expected. In addition, the maximum RSS memory usage is higher than the final mesh size because the generation algorithms also take up memory while running.

2.3 Mesh regularization

If we use random points to generate the mesh, there will be, just by chance, cells that are way smaller than the average cell. As we will see later on, for stability reasons, an upper limit for the simulation timestep is given by the smallest cells. If the smallest cells were way smaller than the average cell, the timestep would be really small and we would have to make too many simulation steps for the average resolution. Therefore, we want the cells to be roughly the same size so that the timestep can be as large as possible for the average resolution. A way to do this is to relax the mesh. This can be done, for example, using Lloyds algorithm. The idea of this algorithm is to generate the mesh using the seedpoints and then calculate the centroid of each cell (Springel, 2010). The centroid $\vec{C}$ is calculated using

$$C_x = \frac{-1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{2.1}$$

$$C_y = \frac{-1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \tag{2.2}$$

where A is the cell area, the (x_i, y_i) are the vertices of the cell and the sum is over all N vertices. Those centroids will be used in the next iteration as the new seedpoints. After a few iterations, this will produce close-to-centroidal Voronoi meshes (meaning that the

seed and centroid will be almost the same). In Fig. 18, a mesh with random seedpoints is relaxed using Lloyds algorithm. After 15 iterations, the mesh already looks close to centroidal.

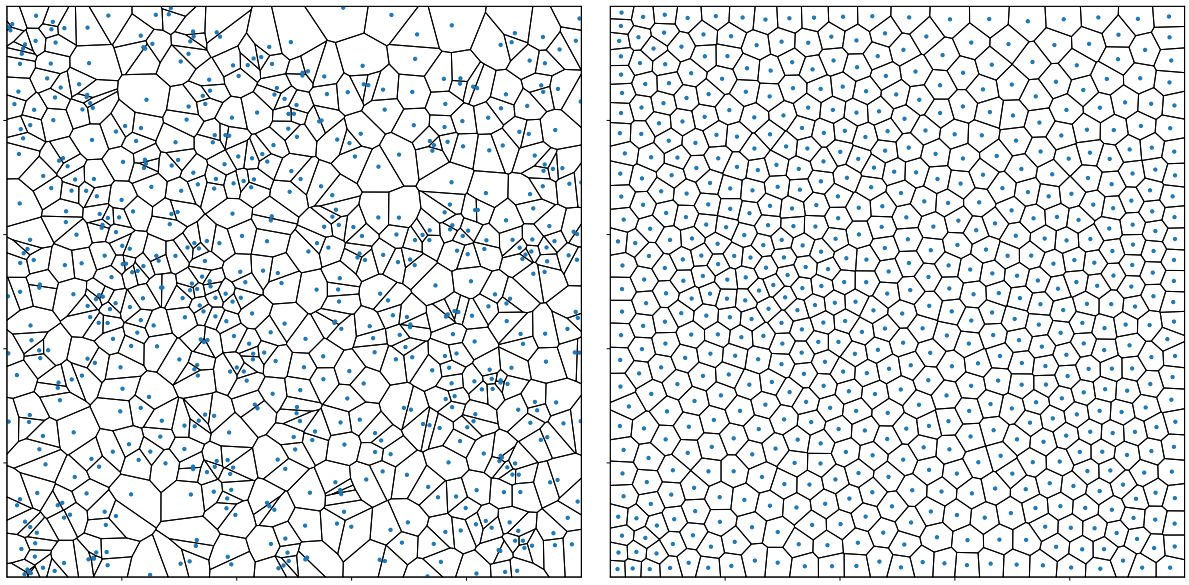

Figure 18: Left: unrelaxed mesh. Right: mesh after 15 Lloyd steps. (Video)

2.4 Internal unstructured boundaries

Since we have a working Voronoi mesh, let's add internal unstructured boundaries following (Springel, 2010). We can use this to model the flow around arbitrary shapes later.
If we want to turn a cell into a boundary cell, we can just set the `is_boundary` boolean to `true` for all its edges. As a result, we will have a cell with no flux through any of the edges. For custom unstructured boundaries, the only thing left to do is to add seedpoints to the random seeds in a way, that the grid will include a custom boundary. If we have a list of vertices of our custom boundary, we can get the correct seeds to insert by placing two seeds just in the middle between two vertices. The seeds are separated perpendicularly to these vertices by a small amount. We then need to remove the random seeds inside of the structure and the seeds outside that are too close. Finally, we set the inner seeds to boundary cells and have a custom internal boundary. The algorithm generally works well. However, the safety distance to outside seeds is really problem-specific. One can also add further boundary cells inside that custom boundary to avoid degeneracies. In Fig. 19, one can see an example circular internal boundary and one shaped like an airfoil.

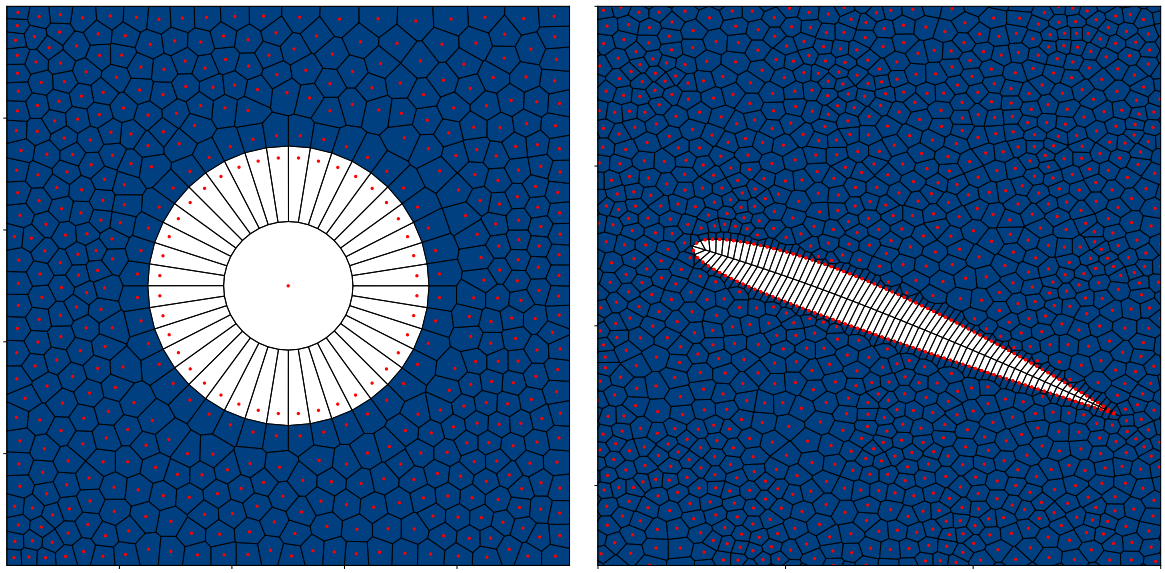

Figure 19: In blue are the simulation cells and in white are the boundary cells. The seedpoints are red. Left: circular internal boundary. Right: internal boundary shaped like an airfoil.

Recap: The Mesh

In this section, we introduced the Mesh structure, which we will use in the following sections, to solve the hydrodynamic equations. For that, we implemented different Voronoi mesh generation algorithms in 2D and demonstrated optimal $\mathcal{O}(N \log N)$ scaling for the faster point insertion algorithm when presorting the seedpoints. After that, we discussed an option to relax the mesh to an almost centroidal mesh and introduced a method to implement unstructured boundaries.

3 Finite Volume Methods

Finite volume methods (FV) divide the space into finite volumes called cells. These store cell averages of the quantities to simulate. The equations are then solved by computing fluxes, based on which the cell averages are updated (Springel et al., 2020). We start by deriving the general finite volume scheme and look at different aspects of implementing it. Then we look at the implementation results of advection, shallow water and Euler equations.

3.1 General derivation

As mentioned before, a general hyperbolic PDE in 2D can be written as

$$\frac{\partial \mathbf{U}}{\partial t} + \vec{\nabla}\vec{\mathbf{F}} = 0, \tag{3.1}$$

where $\mathbf{U}(\vec{x}, t)$ is the state vector and $\vec{\mathbf{F}}(\vec{x}, t) = (\mathbf{F_x}, \mathbf{F_y})$ the flux vector (Springel et al., 2020). To get the finite volume scheme, we start by integrating the conservation equation over a cell C_i.

$$\int_{C_i} \frac{\partial \mathbf{U}}{\partial t} + \vec{\nabla}\vec{\mathbf{F}} dA = 0 \tag{3.2}$$

We then pull out the derivative and separate the integral.

$$\frac{\partial}{\partial t} \int_{C_i} \mathbf{U} dA + \int_{C_i} \vec{\nabla}\vec{\mathbf{F}} dA = 0 \tag{3.3}$$

If we define the cell average of a cell with area A as

$$\langle \mathbf{U} \rangle_i \ = \frac{1}{A} \int_{C_i} \mathbf{U} dA, \tag{3.4}$$

we can substitute it into Eq. (3.3).

$$\frac{\partial}{\partial t} A \langle \mathbf{U} \rangle_i + \int_{C_i} \vec{\nabla}\vec{\mathbf{F}} dA = 0 \tag{3.5}$$

Using Gauss's theorem, we can convert the integral over the flux divergence into a line integral

$$\frac{\partial}{\partial t} A \langle \mathbf{U} \rangle_i + \int_{\partial C_i} \vec{\mathbf{F}} d\vec{n} = 0, \tag{3.6}$$

where $\vec{n}$ is the outward normal on the edge of the cell. So far, this equation is still exact and holds for arbitrary cells (Springel et al., 2020). We now replace the flux by a numerical flux $\vec{\mathbf{F}}^\star$, which approximates the real flux at the edge (see Sect. 3.1.1). Additionally, we let our cell consist of a finite number of edges $j \in \partial C_i$ and assume that the numerical flux is constant along an edge, which reduces the line integral to a sum.

$$\frac{\partial}{\partial t} A \langle \mathbf{U} \rangle_i + \sum_{j \in \partial C_i} \vec{\mathbf{F}}^\star_j \cdot \vec{n_j} \cdot l_j = 0 \tag{3.7}$$

The numerical flux $\vec{\mathbf{F}}_j^\star$ through edge j is a function of the states on both sides of the edge. We will call them $\mathbf{U}_L$ and $\mathbf{U}_R$. For now, these are only the cell averages $\langle \mathbf{U} \rangle_i$ of cell i and $\langle \mathbf{U} \rangle_j$ of the neighboring cell, which makes this a piecewise constant approximation (see Fig. 20). Later, we will improve this estimate to achieve higher-order accuracy. Now, the only thing left to do is to discretize in time. The cell average $\langle \mathbf{U} \rangle_i$ at the timestep n will be called $\mathbf{U}_i^n$ and we employ a simple forward Euler scheme.

$$A \frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^n}{\Delta t} + \sum_{j \in \partial C_i} \vec{\mathbf{F}}_j^\star(\mathbf{U}_L^n, \mathbf{U}_R^n) \cdot \vec{n_j} \cdot l_j = 0 \tag{3.8}$$

When solving for $\mathbf{U}_i^{n+1}$, we get the finite volume update scheme (Springel et al., 2020).

$$\boxed{\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \frac{\Delta t}{A} \sum_{j \in \partial C_i} \vec{\mathbf{F}}_j^\star(\mathbf{U}_L^n, \mathbf{U}_R^n) \cdot \vec{n_j} \cdot l_j} \tag{3.9}$$

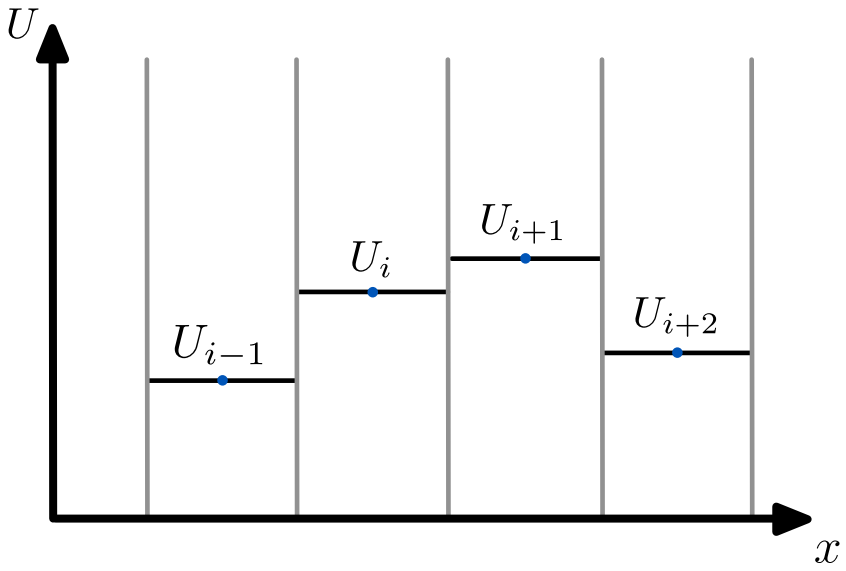

Figure 20: Representation of a cell quantitiy U in FV. Using the cell average leads to a piece-wise constant approximation. (Springel et al., 2020)

3.1.1 Flux estimates

To use the FV update scheme, we need to choose the numerical flux $\vec{\mathbf{F}}_j^\star(\mathbf{U}_L^n, \mathbf{U}_R^n)$. Given that it will be the combination of $\mathbf{U}_L^n$ and $\mathbf{U}_R^n$, a naive guess is to choose the average of both for the flux estimate.

$$\vec{\mathbf{F}}_j^\star = \vec{\mathbf{F}}\left(\frac{\mathbf{U}_L^n + \mathbf{U}_R^n}{2}\right) \tag{3.10}$$

This, however, turns out to be completely numerically unstable (Springel et al., 2020). The problem is that we have not properly accounted for the characteristics of the system. As we have seen before, the characteristics tell us how information (e.g. a small perturbation) propagates through the system. Using the average flux of both states is equivalent to equally taking information from the upwind and downwind sides. This leads to problems for advection since there should be no information coming from the downwind side by definition. (Springel et al., 2020)

Upwind
Therefore, the first flux estimate for advection is to only use information from the upwind side.

$$\vec{\mathbf{F}}_j^\star = \begin{cases} \vec{\mathbf{F}}(\mathbf{U}_L^n) & \text{if } \mathbf{U}_L^n \text{ is upwind} \\ \vec{\mathbf{F}}(\mathbf{U}_R^n) & \text{if } \mathbf{U}_R^n \text{ is upwind} \end{cases} \tag{3.11}$$

We will call this an upwind scheme. (Springel et al., 2020)

Lax Friedrichs
Alternatively, one can still use the average estimate but with an added diffusion term to stabilize.

$$\vec{\mathbf{F}}_j^\star = \vec{\mathbf{F}}\left(\frac{\mathbf{U}_L^n + \mathbf{U}_R^n}{2} - \frac{\Delta x_{ij}}{2\Delta t}(\mathbf{U}_R^n - \mathbf{U}_L^n)\right) \tag{3.12}$$

Here Δx_{ij} is the distance between the seeds of cell i and j (Knezevic, 2012). On the one hand, this scheme has the advantage that one does not care about the characteristics. On the other hand, it has the disadvantage of adding more diffusion than necessary.

Riemann Problem
If we want to consider the characteristics of more complex equations (e.g. shallow water or Euler), the upwind equivalent is far less obvious. The idea here is to solve Riemann problems face-wise and use the solution of the Riemann flux in the updating step (Springel et al., 2020). This idea dates back to Godunov, which is the reason why such schemes are often called Godunov schemes (Godunov, 1959).
The Riemann problem is an initial value problem for $t = 0$ consisting of two spatially constant states $\mathbf{U}_L$, $\mathbf{U}_R$ separated by a plane. For shallow water, this is

$$\mathbf{U}_L = \begin{pmatrix} h_L \\ h_L u_L \\ h_L v_L \end{pmatrix}, \qquad \mathbf{U}_R = \begin{pmatrix} h_R \\ h_R u_R \\ h_R v_R \end{pmatrix} \tag{3.13}$$

and for Euler equations, it is

$$\mathbf{U}_L = \begin{pmatrix} \rho_L \\ \rho_L u_L \\ \rho_L v_L \\ E_L \end{pmatrix}, \qquad \mathbf{U}_R = \begin{pmatrix} \rho_R \\ \rho_R u_R \\ \rho_R v_R \\ E_R \end{pmatrix}. \tag{3.14}$$

The goal is to get the evolution for $t > 0$ to get the flux through the plane, which will be our cell face. It is sufficient to look at the 1D Riemann problem since one can always rotate into the coordinate frame of the cell face, solve the 1D problem there and rotate back (Springel, 2010). For a Riemann problem, exact solutions exist apart from the need to find a root numerically. The solution contains three wave types traveling with speeds defined by different characteristics (see Fig. 21). The middle wave is a contact discontinuity marking the boundary between the original two states. It is enclosed by a shock or a rarefication wave on either side (Springel et al., 2020).

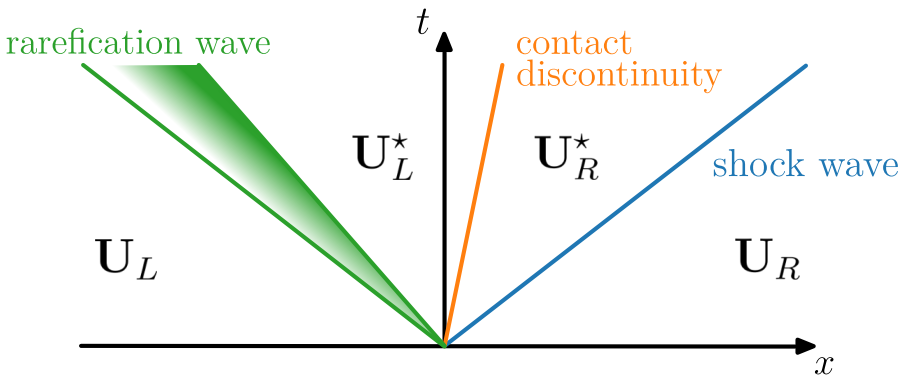

Figure 21: Characteristics of a Riemann problem.

Between those waves, the states are constant in time. Fig. 22 shows a solution to the Riemann problem at $t = 0.2$ for Euler equations.

The special case where the initial velocity of the Riemann problem is zero is called dam break for shallow water equations and Sod's shock tube for Euler equations. We will use those special cases to verify our algorithm later on.

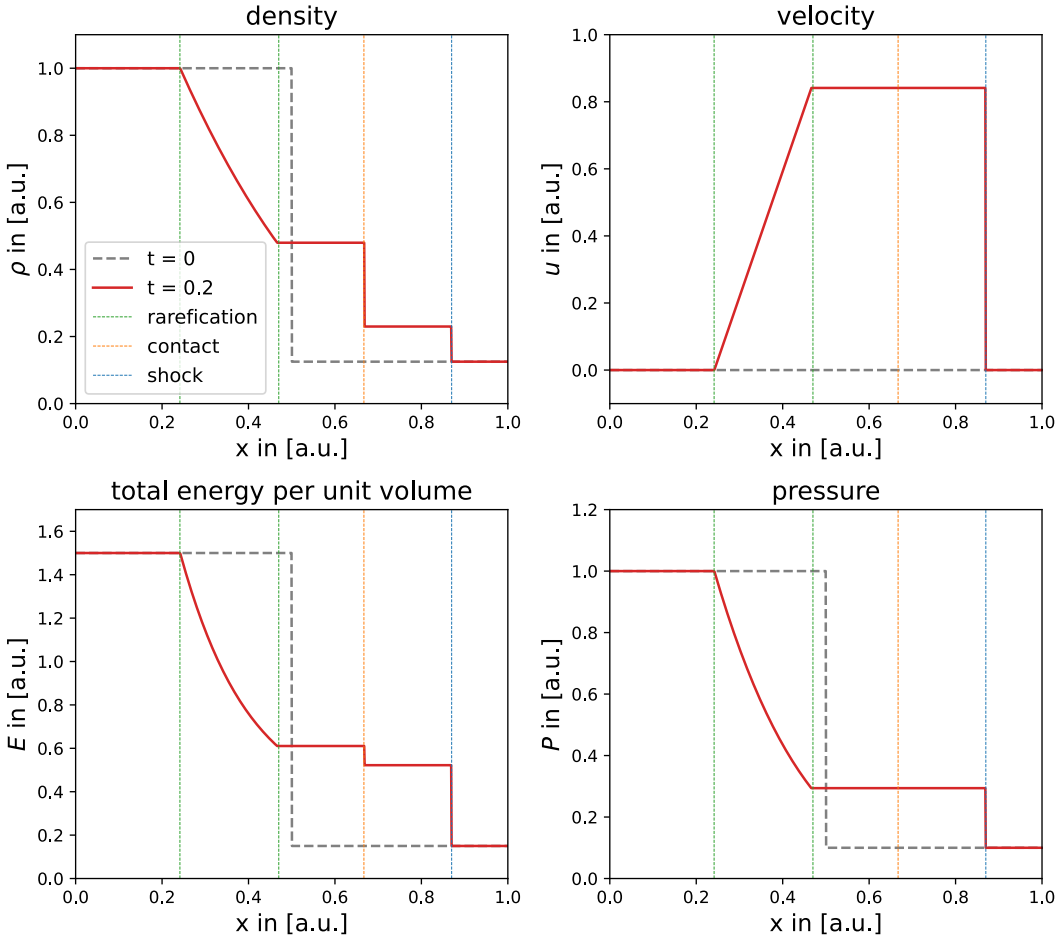

Figure 22: Solution to a Riemann problem at $t = 0.2$. The initial condition here is without velocity. Hence this is the special case of a Sod's shock tube.

HLL approximate Riemann solver

The approximate HLL (Harten-Lax-van Leer) Riemann solver ignores the middle contact discontinuity and decomposes the problem into two waves, which are the forward- and backward-moving sound waves. To solve the 2D flux through the cell faces, we rotate the flux into the frame of the face to effectively solve a 1D problem. In 1D, the speed of the moving sound waves is given by

$$S_L = \min\left(u_L - c_L,\ u_R - c_R\right) \tag{3.15}$$

$$S_R = \max\left(u_L + c_L,\ u_R + c_R\right), \tag{3.16}$$

where the sound speed for shallow water equations is given by $c_i = \sqrt{gh_i}$ and for Euler equations by $c_i = \sqrt{\gamma \frac{P_i}{\rho_i}}$. The flux through the face is then estimated as

$$\mathbf{F}_{HLL} = \begin{cases} \mathbf{F}_L & \text{if } S_L \geq 0 \\ \frac{S_R \mathbf{F}_L - S_L \mathbf{F}_R + S_L S_R (\mathbf{U_R} - \mathbf{U}_L)}{S_R - S_L} & \text{if } S_L < 0 < S_R \\ \mathbf{F}_R & \text{if } S_R \leq 0 \end{cases} . \tag{3.17}$$

After rotating back into the global frame, we can then use this flux estimate for $\vec{\mathbf{F}}^\star$ (Dullemond and H.H.Wang, 2009). The main disadvantage of the HLL solver is that it cannot keep contact discontinuities sharp. This is not surprising since it ignores the middle wave. More advanced Riemann solvers exist, like the HLLC solver, where the C stands for including the contact discontinuity (Dullemond and H.H.Wang, 2009). However, they are not implemented here.

3.1.2 Stability

Apart from correctly estimating the fluxes, we must also ensure the time integration is stable. For that, we need to make the timestep sufficiently small. There is an upper limit to Δt when the integration breaks down. We can think of the maximum timestep in terms of information travel. If a timestep is large enough that a characteristic starting from the neighbor of a neighbor cell is able to reach our cell, one needs more than the direct neighbor to calculate the flux of that timestep accurately (see Fig. 23). Since we will not do that in our scheme, it will break down for such large timesteps. We, therefore, have to obey the so-called Courant-Friedrichs-Levy (CFL) timestep condition

$$\Delta t \leq \frac{\Delta x}{c}, \tag{3.18}$$

where Δx is the distance between two seeds and c is the speed of the fastest characteristic (e.g. for advection $c = v$) (Springel et al., 2020). Note that this condition is necessary but not sufficient for all cases. Later in the simulations, we will always fulfill the CFL condition with some margin. However, choosing small fudge factors is also undesirable since the simulation time grows antiproportional to it. We, therefore, use fudge factors like $0.2 \cdot \Delta t$ instead of Δt or similar to balance simulation time and stability.

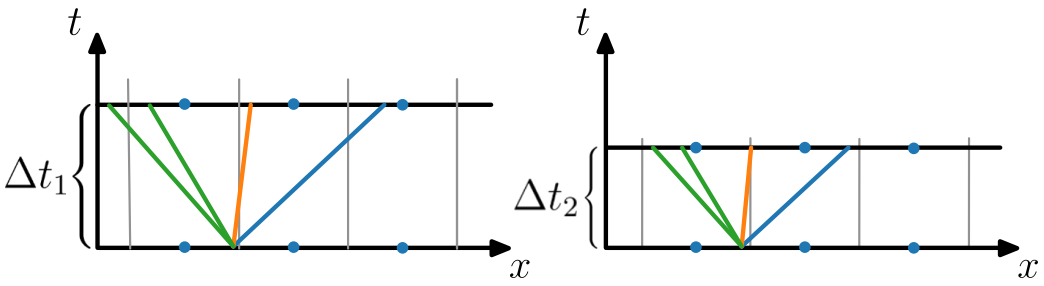

Figure 23: Left: CFL condition not fulfilled. Information flows more than one cell per timestep. Right: CFL condition fulfilled. Information flows less than one cell per timestep.

3.1.3 Boundary conditions

To fully define the simulation, we need to specify boundary conditions. These will be applied to all faces where the `is_boundary` boolean is set to `true`. All the internal unstructured boundaries, therefore, also obey these conditions. We implement the boundary conditions by setting the neighbor state of such faces to the correct state.

Inflow, outflow
For advection the entire flow is towards one direction. We will, therefore, have faces, where we specify an outflow condition and faces, where we specify an inflow condition. For the flow out of the mesh, we can simply set the neighbor's state to the same as the cell's state U_i.

$$U_{out} = U_i \tag{3.19}$$

In principle, the flow into the mesh can be set to arbitrary inflow conditions. We, however, will set it to

$$U_{in} = 0, \tag{3.20}$$

such that the inflow is zero (Knezevic, 2012).

For shallow water equations and Euler equations, we again realize the boundary conditions by setting the correct neighbor state.

Zero gradient
A perfect outflow condition for these equations is hard to realize. A simple approximation to that is to just set the gradient at the boundaries to zero. This effectively means setting the neighbor's state to the same as the cell's state.

$$\mathbf{U}_j = \mathbf{U}_i \tag{3.21}$$

Reflective
For reflective boundary conditions, we want to cancel any flow through the face. We achieve that by setting the neighbor's state to the cell's state but inverting the velocity component normal to the face (see Fig. 24). This cancels the momentum flow and sets all other quantities to zero gradient.

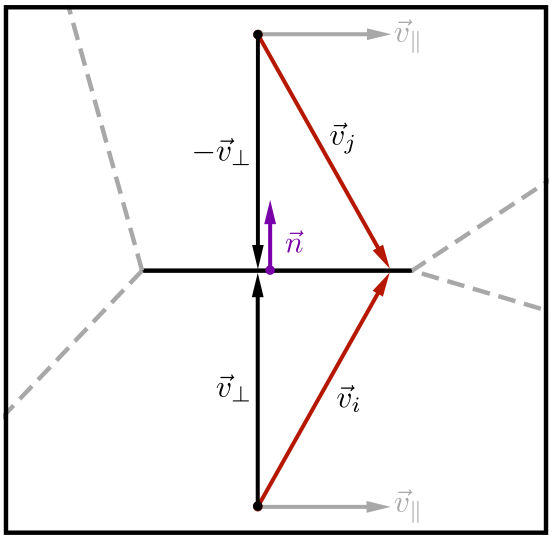

Figure 24: Velocity reflection of the component normal to the face.

The neighbor velocity is given by

$$\vec{v}_j = \vec{v}_i - 2\vec{v}_{\perp} = \vec{v}_i - 2(\vec{v}_i \cdot \vec{n}) \cdot \vec{n}, \tag{3.22}$$

where $\vec{n}$ is the normal vector of the face. For shallow water, this looks like

$$\mathbf{U}_j = \begin{pmatrix} h_j \\ h_j u_j \\ h_j v_j \end{pmatrix} = \begin{pmatrix} h_i \\ h_i(u_i - 2\,(u_i \cdot n_x + v_i \cdot n_y) \cdot n_x) \\ h_i(v_i - 2\,(u_i \cdot n_x + v_i \cdot n_y) \cdot n_y) \end{pmatrix} \tag{3.23}$$

and for Euler equations like

$$\mathbf{U}_j = \begin{pmatrix} \rho_j \\ \rho_j u_j \\ \rho_j v_j \\ E_j \end{pmatrix} = \begin{pmatrix} \rho_i \\ \rho_i(u_i - 2\,(u_i \cdot n_x + v_i \cdot n_y) \cdot n_x) \\ \rho_i(v_i - 2\,(u_i \cdot n_x + v_i \cdot n_y) \cdot n_y) \\ E_i \end{pmatrix}. \tag{3.24}$$

Periodic

The final boundary condition we are looking at is a periodic boundary. For the Cartesian mesh, we achieve periodic boundaries by setting the `is_boundary` boolean to `false` for all boundary faces. We then set the neighbor pointers of the boundary faces to the correct neighbors on the other side.

For Voronoi, we do something similar but have to make sure that the mesh on the bottom fits into the mesh on the top, as well as the mesh from the left fits into the mesh on the right (Springel, 2010). We do this, in a very naive way, by generating a mesh with nine times the same pointset arranged in a 3x3 grid (see. Fig. 25). The middle grid will be the mesh used later in the simulation and is stored first in the array. After generating the entire mesh, we set the neighbor pointers by taking the existing neighbor relations and replacing them with the corresponding ones in the middle mesh. This is done by a simple modulo operation, taking the current neighbor index modulo the total number of the middle seedpoints. This returns the correct neighbor index inside of the middle mesh. After that, we delete all the cells, apart from the middle mesh, and have a finished periodic boundary mesh.

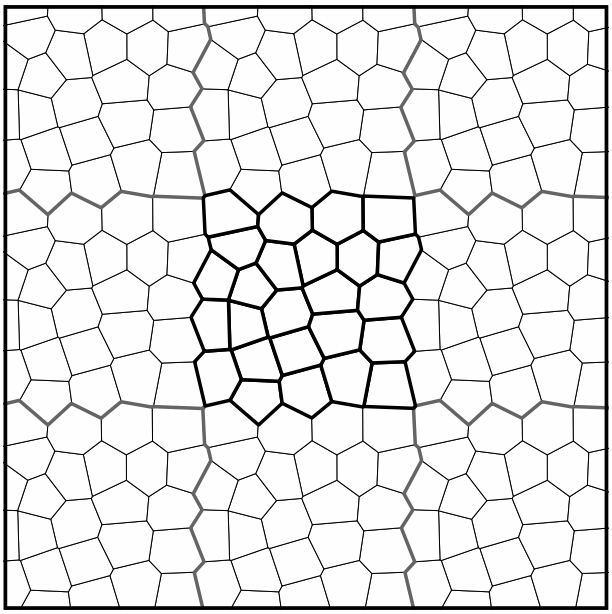

Figure 25: Nine times the original mesh on a 3x3 grid. The middle grid will be the mesh with periodic boundary conditions.

3.1.4 Second-order: MUSCL scheme

While the discussed first-order finite volume method is robust and simple to implement, it suffers from excessive numerical diffusion and only has first-order convergence (Springel et al., 2020). By the convergence order of a scheme, we mean the convergence rate of the error in smooth regions of the flow. We can quantify the error through the L1 error norm

$$L1 = \frac{1}{N} \sum_{i=1}^{N} |U_i - U(x_i)|, \tag{3.25}$$

which is the average error per cell. In a working scheme, this error decreases with N and converges to the true solution. With sufficient computational resources, one could then get below any error level. Since we do not have infinite computational resources, the convergence rate is crucial to achieve accurate simulations. First-order convergence means doubling the resolution halves the L1 error, while second-order convergence reduces the L1 error by a factor of four. It is possible to construct even higher-order schemes, but at least for finite volume, they tend to become much more complex quickly, so there eventually is a point of diminishing return (Springel et al., 2020). Implementing second-order is straightforward and drastically improves the accuracy at a given resolution or compute time. To construct a second-order method, we follow the MUSCL-Hancock scheme (van Leer, 1984), (Toro, 1997), (van Leer, 2006) and replace the piece-wise constant approximation (as in Fig. 20) with a piece-wise linear reconstruction (see. Fig. 26). For that, we first need to estimate the gradient for each cell. These are then slope-limited if needed, such that the extrapolation does not induce new extrema. The linearly extrapolated states are then used to solve the face-wise Riemann problems. Based on those, we update the mesh. (Springel et al., 2020)

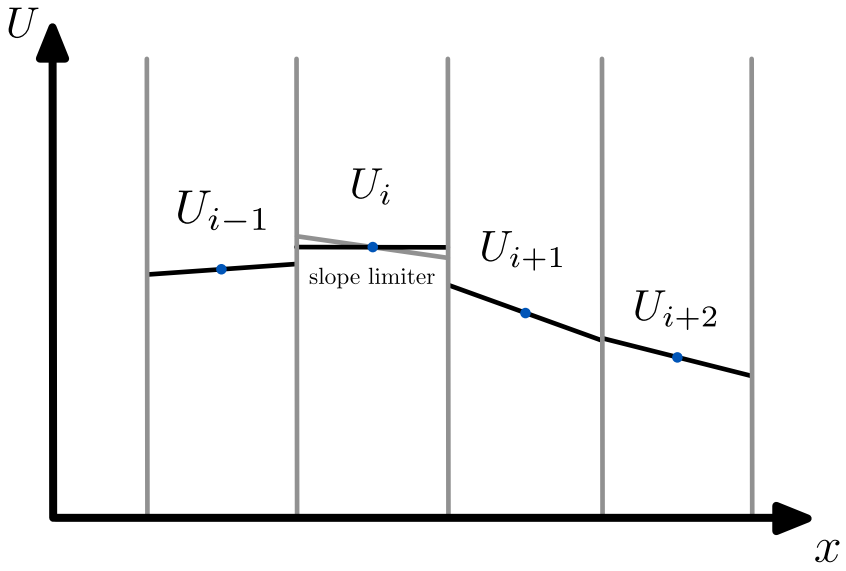

Figure 26: Cell representation in second-order MUSCL scheme. Piece-wise linear approximation plus additional slope limiting. (Springel et al., 2020)

Gradient calculation

To calculate the mean gradient $\langle\vec{\nabla}\phi\rangle_i = (\langle\partial_x\phi\rangle_i, \langle\partial_y\phi\rangle_i)$ of a quantity $\phi_i \in \mathbf{U}_i$ (like ρ_i or v_i) we employ Gauss's theorem and approximate both sides:

$$A_i \cdot \langle\vec{\nabla}\phi\rangle_i \approx \int_{C_i} \vec{\nabla}\phi dA = \int_{\partial C_i} \phi d\vec{n} \approx \sum_j \frac{\phi_i + \phi_j}{2} l_j \vec{n}. \tag{3.26}$$

Here A_i is the area of Cell C_i, $\vec{n}$ is the vector normal to the face, l_j is the length of face j and we sum over all the neighbors j. This gradient approximation can be optimized for the special case of a Voronoi tesselation. We will follow the optimization of (Springel, 2010) for that and use the formula

$$\langle\vec{\nabla}\phi\rangle_i = \frac{1}{A_i}\sum_j l_j\Big([\phi_j - \phi_i]\frac{\vec{c}_{ij}}{r_{ij}} - \frac{\phi_i + \phi_j}{2}\vec{n}\Big), \tag{3.27}$$

where $\vec{c}_{ij}$ is the vector from the mid-point between the seed positions r_i and r_j to the center of the corresponding edge and r_{ij} is the distance between r_i and r_j. This gradient is exact to linear order. (Springel, 2010), (Schaal, 2013)

Slope limiting

Before extrapolating, we need to slope limit the gradients. This is an important step to prevent artificial minima and maxima from being induced. Otherwise, oscillations near discontinuities occur and grow to become numerically unstable (see results section later on).

We slope limit the gradients by a local factor $\alpha_i \in [0, 1]$, that we need to calculate for each cell.

$$\langle\vec{\nabla}\phi\rangle_i \to \alpha_i\langle\vec{\nabla}\phi\rangle \tag{3.28}$$

The first slope limiter implemented follows (Springel, 2010). The α_i here ensures that the reconstructed quantities at the center of an edge are not larger than the maximum ϕ_i^{max} and not smaller than the minimum ϕ_i^{min} among all neighboring states and the state itself. It is calculated using:

$$\alpha_i = \min_{j \in \partial C_i} (1, \psi_{ij}) \tag{3.29}$$

$$\psi_{ij} = \begin{cases} (\phi_i^{max} - \phi_i)/\Delta\phi_{ij} & \text{for } \Delta\phi_{ij} > 0 \\ (\phi_i^{min} - \phi_i)/\Delta\phi_{ij} & \text{for } \Delta\phi_{ij} < 0 \\ 1 & \text{for } \Delta\phi_{ij} = 0 \end{cases} \tag{3.30}$$

with $\Delta\phi_{ij} = (\vec{f}_{ij} - \vec{s}_i)$, where $\vec{f}_{ij}$ is the center of the edge and $\vec{s}_i$ the seed of cell i. The downside of this slope limiter is that it is not total variation diminishing (TVD), which means that it can cause small post-shock oscillations.
We implemented a second slope limiter, which is TVD, but therefore also more diffusive. It follows (Duffell and MacFadyen, 2011) and is given by

$$\psi_{ij} = \begin{cases} \max\left(\theta\left(\phi_j - \phi_i\right)/\Delta_{ij}, 0\right) & \text{for } \Delta\phi_{ij} > 0 \\ \max\left(\theta\left(\phi_j - \phi_i\right)/\Delta_{ij}, 0\right) & \text{for } \Delta\phi_{ij} < 0\,, \\ 1 & \text{for } \Delta\phi_{ij} = 0 \end{cases} \tag{3.31}$$

with θ controlling the diffusivity and generally being set to $\theta = 1$. (Schaal, 2013)

Extrapolation
With the slope limited gradients, we can estimate the left and right states adjacent to the face using spatial linear extrapolation. For $\mathbf{U}^L$, the extrapolation starts from the current cell C_i while the extrapolation for $\mathbf{U}^R$ starts from the neighbor cell C_j.

$$\mathbf{U}^L = \mathbf{U}_i + \langle \partial_x \mathbf{U} \rangle_i \frac{\Delta x_i}{2} + \langle \partial_y \mathbf{U} \rangle_i \frac{\Delta y_i}{2} \tag{3.32}$$

$$\mathbf{U}^R = \mathbf{U}_j + \langle \partial_x \mathbf{U} \rangle_j \frac{\Delta x_j}{2} + \langle \partial_y \mathbf{U} \rangle_j \frac{\Delta y_j}{2} \tag{3.33}$$

Here $\left(\frac{\Delta x}{2}, \frac{\Delta y}{2}\right)$ is the distance from the center of the cell to the middle of the face. For a Cartesian mesh, this is just half the cell width.
These states could then be used in the flux solver, where we will ignore that our reconstruction now has a gradient over the cell and still solve face-wise Riemann problems (Springel et al., 2020). However, we also need to consider that our quantities are functions of space and time. The total differential, therefore, obtains an additional contribution from varying in time.

$$d\mathbf{U}(x, y, t) = \frac{\partial \mathbf{U}}{\partial x} dx + \frac{\partial \mathbf{U}}{\partial y} dy + \frac{\partial \mathbf{U}}{\partial t} dt \tag{3.34}$$

Including this in our scheme is necessary to reach second-order accuracy in time and for stability reasons (Springel et al., 2020). We, therefore, need to extrapolate in space and time.

$$\mathbf{U}^L = \mathbf{U}_i + \langle \partial_x \mathbf{U} \rangle_i \frac{\Delta x_i}{2} + \langle \partial_y \mathbf{U} \rangle_i \frac{\Delta y_i}{2} + \langle \partial_t \mathbf{U} \rangle_i \frac{\Delta t}{2} \tag{3.35}$$

$$\mathbf{U}^R = \mathbf{U}_j + \langle \partial_x \mathbf{U} \rangle_j \frac{\Delta x_j}{2} + \langle \partial_y \mathbf{U} \rangle_j \frac{\Delta y_j}{2} + \langle \partial_t \mathbf{U} \rangle_j \frac{\Delta t}{2} \tag{3.36}$$

We have an estimate for the spatial gradient. To get an estimate of the time derivative, we use the general hyperbolic PDE form in state vector notation (see Eq. (1.7)) to write the time derivative in terms of the flux Jacobians and the gradient.

$$\partial_t \mathbf{U} = -\vec{\nabla} \cdot \vec{\mathbf{F}} = -\left(\frac{\partial \mathbf{F_x}}{\partial \mathbf{U}} \cdot \partial_x \mathbf{U} + \frac{\partial \mathbf{F_y}}{\partial \mathbf{U}} \cdot \partial_y \mathbf{U} \right) \tag{3.37}$$

In the discrete version, this becomes

$$\langle \partial_t \mathbf{U} \rangle_i = -\left(\left. \frac{\partial \mathbf{F_x}}{\partial \mathbf{U}} \right|_{\mathbf{U}_i} \cdot \langle \partial_x \mathbf{U} \rangle_i + \left. \frac{\partial \mathbf{F_y}}{\partial \mathbf{U}} \right|_{\mathbf{U_i}} \cdot \langle \partial_y \mathbf{U} \rangle_i \right). \tag{3.38}$$

We can substitute this into the extrapolation and get:

$$\mathbf{U}^L = \mathbf{U}_i + \left(\frac{\Delta x_i}{2} \mathbb{1} - \frac{\Delta t}{2} \left. \frac{\partial \mathbf{F}_x}{\partial \mathbf{U}} \right|_{\mathbf{U}_i} \right) \langle \partial_x \mathbf{U} \rangle_i + \left(\frac{\Delta y_i}{2} \mathbb{1} - \frac{\Delta t}{2} \left. \frac{\partial \mathbf{F}_y}{\partial \mathbf{U}} \right|_{\mathbf{U}_i} \right) \langle \partial_y \mathbf{U} \rangle_i \tag{3.39}$$

$$\mathbf{U}^R = \mathbf{U}_j + \left(\frac{\Delta x_j}{2} \mathbb{1} - \frac{\Delta t}{2} \left. \frac{\partial \mathbf{F}_x}{\partial \mathbf{U}} \right|_{\mathbf{U}_j} \right) \langle \partial_x \mathbf{U} \rangle_j + \left(\frac{\Delta y_j}{2} \mathbb{1} - \frac{\Delta t}{2} \left. \frac{\partial \mathbf{F}_y}{\partial \mathbf{U}} \right|_{\mathbf{U}_j} \right) \langle \partial_y \mathbf{U} \rangle_j. \tag{3.40}$$

Using these extrapolated values for solving the face-wise Riemann problems defines the MUSCL-Hancock scheme (Springel et al., 2020), which is a second-order accurate extension of Godunov's method (our first-order FV update scheme with the Riemann Solver). Even higher-order extensions to Godunov's method exist and reconstruct higher-order functions on the grid (e.g. piece-wise parabolic method (PPM)). Depending on the order, many more cells in the environment need to be considered to determine the reconstruction coefficients. This, for example, can be done using a least square fitting procedure. (Springel et al., 2020)

3.2 Implementation and Results

The implemented version of the theory discussed above includes an upwind scheme for advection and a first- and second-order Godunov scheme with HLL solver for the SWE and Euler equations. We will now test the implementation by comparing it to analytically known solutions and looking at further test problems.

3.2.1 Advection

We start by looking at the upwind scheme for scalar advection. Here the analytic solution to any initial condition $u(\vec{x}, 0)$ is simply $u(\vec{x}, t) = u(\vec{x} - \vec{v}t, 0)$. In this paragraph, we perform different simulations to test this scheme (see Tab. 2).

Dim.	Initial cond.	N	Mesh	Velocity v	Comment
1D	step func	100	Cartesian	0.5	num. diffusion
1D	gaussian	100	Cartesian	0.5	num. diffusion
1D	step func	100, 500, 1000	Cartesian	0.5	resolution study
2D	circle	10000	Cartesian	$(0, 0.5), \frac{1}{\sqrt{2}}(0.5, 0.5)$	angular dependence
2D	circle	10000	Voronoi	$(0, 0.5), \frac{1}{\sqrt{2}}(0.5, 0.5)$	no angular dependence

Table 2: Advection simulations

Numerical diffusion
We first consider the 1D case for an initial step function and an initial Gaussian.

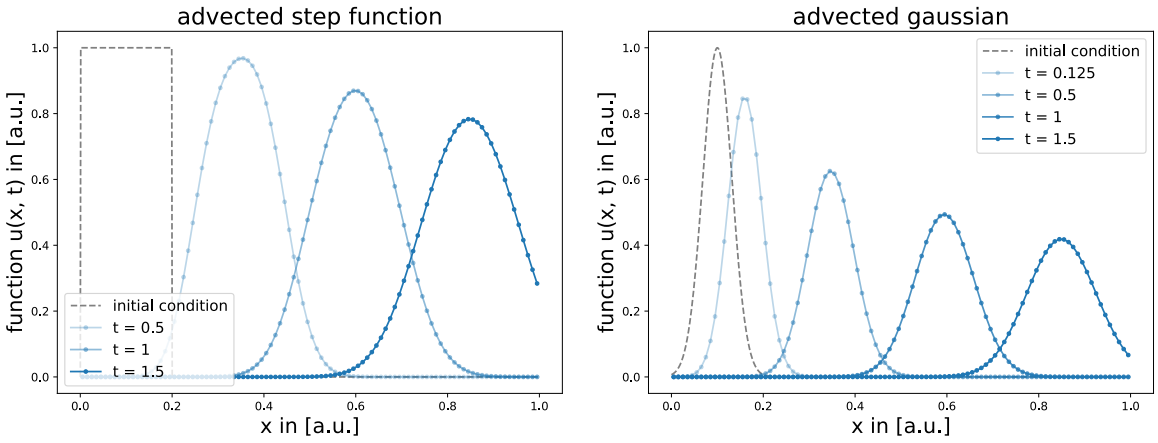

Figure 27: Advected step function and Gaussian for $N = 100$ and $v = 0.5$. The initial condition is advected to the right while diffusing.

Fig. 27 shows that the solution is advected to the right while diffusing. This so-called numerical diffusion is a result of the discretization. In fact, no numerical algorithm exists without numerical diffusion. Some algorithms have more, others less, but a bit of numerical diffusion is unavoidable (Dullemond and H.H.Wang, 2009). The 1D upwind scheme uses the upwind derivative at $i - 1/2$ instead of the derivative at i for stability reasons.

This, of course, is not entirely true, as it should use the i derivative for updating u_i. If we rewrite the derivative at $i - 1/2$

$$\frac{u_i - u_{i-1}}{\Delta x} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} - \frac{\Delta x}{2}\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}, \tag{3.41}$$

we see that it can be rewritten into the centered difference at i and a second term consisting of a constant times the discretized second derivative of u. When plugging that into the discretized version of the advection equation

$$0 = \frac{u_i^{n+1} - u_i^n}{\Delta t} + v \cdot \frac{u_i^n - u_{i-1}^n}{\Delta x} = \frac{u_i^{n+1} - u_i^n}{\Delta t} + v \cdot \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} - \frac{\Delta x v}{2}\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \tag{3.42}$$

that the second term can be recognized as a diffusion term with a diffusion constant

$$D = \frac{\Delta x v}{2}. \tag{3.43}$$

This shows that the upstream difference is equivalent to a centered difference with supplemented diffusion (Dullemond and H.H.Wang, 2009). Of course, this does not account for the entire diffusion since the centered difference is also merely an approximation. Nevertheless, it is useful to think of the upwind scheme effectively solving

$$\partial_t u + v\partial_x u - \frac{\Delta x v}{2}\partial_x^2 u. \tag{3.44}$$

As one can see, the diffusion vanishes for $\Delta x \to 0$. We can confirm this by simulating the step function for different resolutions.

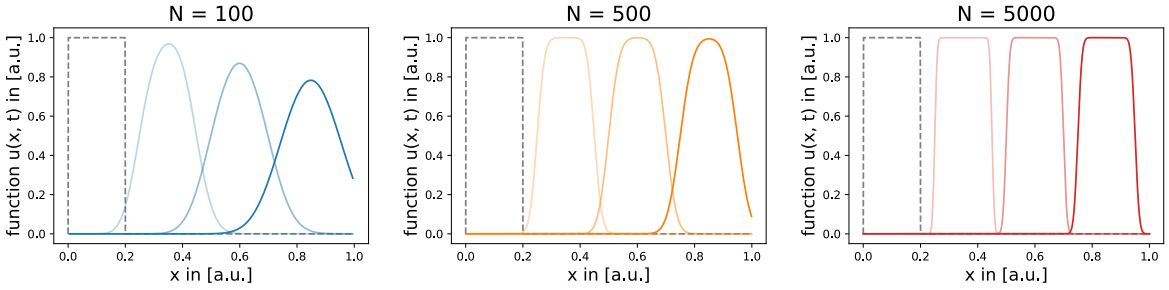

Figure 28: Advected step function for different resolutions $N = 100, 500, 5000$. The higher the resolution, the less numerical diffusion.

In Fig. 28, one clearly sees that the high-resolution versions suffer from far less numerical diffusion. Looking at the L1 error of the step function growing over time, we can see that the error grows roughly proportional to $\sqrt{Dt}$, which is intuitive for diffusion (see Fig. 29). The decreasing L1 error at the end is due to the step function starting to leave the simulation domain and, therefore, contributing less to the error. For the L1 error over N we can use, that

$$\sqrt{Dt} = \sqrt{\frac{\Delta x v}{2}t} = \sqrt{\frac{v}{2N}t} \propto \frac{1}{\sqrt{N}} \tag{3.45}$$

and therefore expect that the error should scale with $1/\sqrt{N}$. As one can see in Fig. 29 the L1 error fit converges with $N^{-0.5}$ verifying the $1/\sqrt{N}$ convergence.

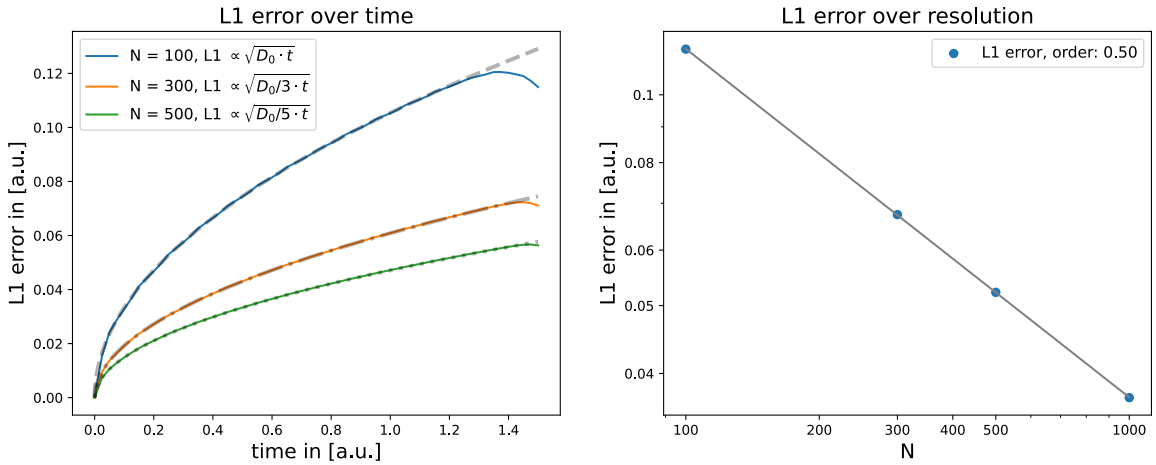

Figure 29: Left: L1 error of step function growing over time with $\sqrt{Dt}$. Right: L1 error of step function decreasing with resolution as $1/\sqrt{N}$.

Conservation

If we look at the total u on the grid over time, we want this to be conserved apart from the outflow. If we look at periodic boundary conditions, there is no outflow, and hence, the total u should be conserved. Looking at the total u evolution over time, one can see in Fig. 30 that the upwind scheme conserves u at machine precision (10^{-16}).

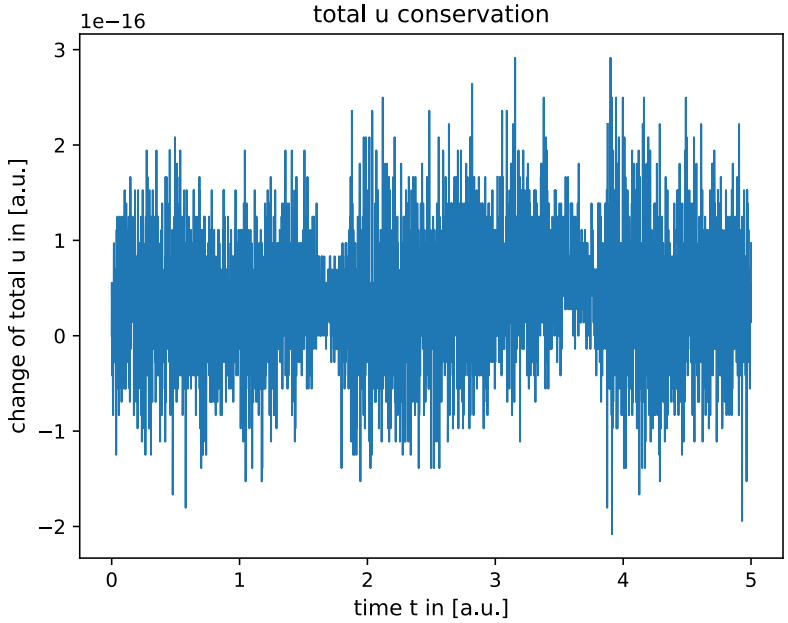

Figure 30: Change of total u over time. Total u varies from the initial total u only by machine precision.

Angular dependence in 2D

We now move into the second dimension and look at the advection of a circle on 2D Cartesian and Voronoi meshes. Here, we again observe numerical diffusion. Because of the intrinsic directions induced by the Cartesian mesh, we observe an angular dependence of the L1 error (see Fig. 31).

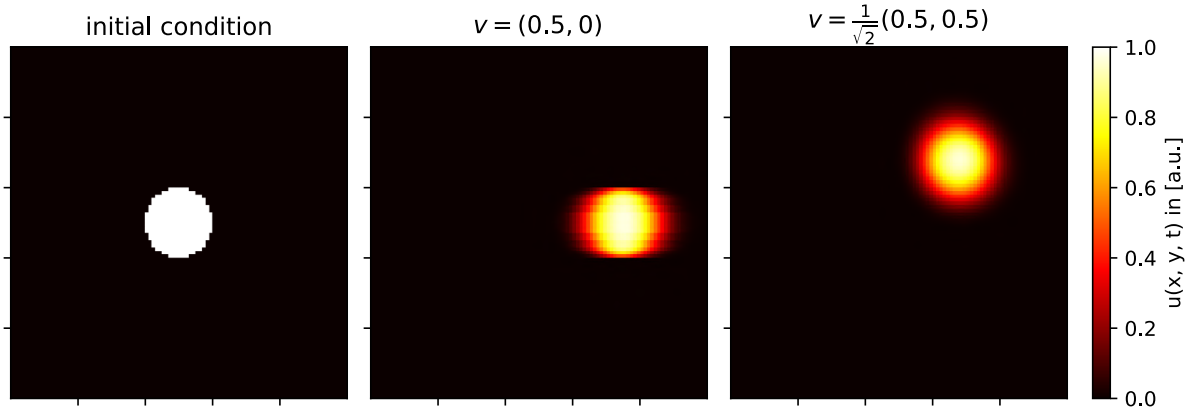

Figure 31: Advected circle for a Cartesian mesh with $N = 10000$ cells. The advection error in the diagonal direction is larger than to the right. (Video)

For the Voronoi mesh, which has no intrinsic direction, this angular dependence does not exist (see Fig. 32).

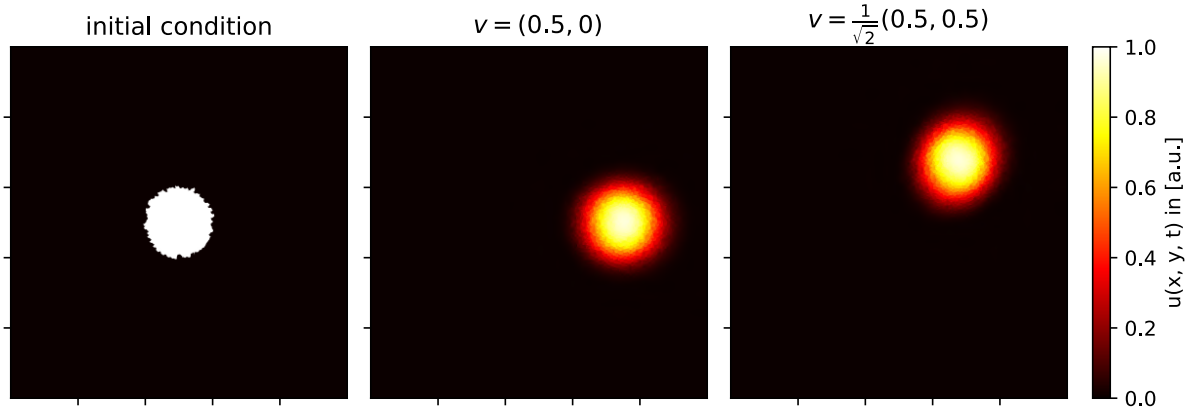

Figure 32: Advected circle on a Voronoi mesh with $N = 10000$ cells . The advection error in the diagonal direction is similar than to the right. (Video)

We can also see this if we directly compare the time evolution of the L1 error in Fig. 33. Here, the two lines differ for Cartesian, while the Voronoi lines are roughly the same. However, the L1 error for Voronoi is equivalent to the Cartesian diagonal error (i.e., the worse error).

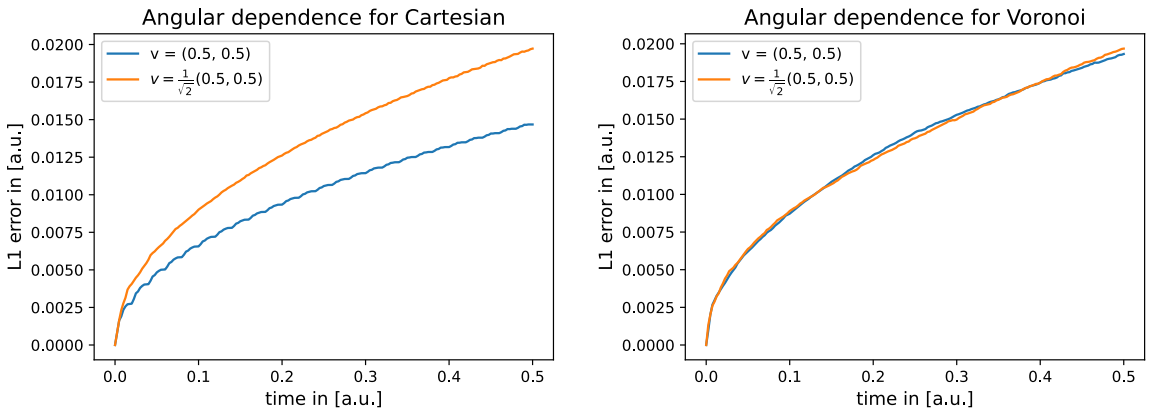

Figure 33: L1 error over time for two different directions. The Cartesian mesh shows an angular dependence, while the Voronoi mesh does not.

Asymmetric solution

Because of the asymmetric way we choose the flux (upwind), the solution is also slightly asymmetric. This is hard to see in a normal plot but is visible using logarithmic scaling (see Fig. 34).

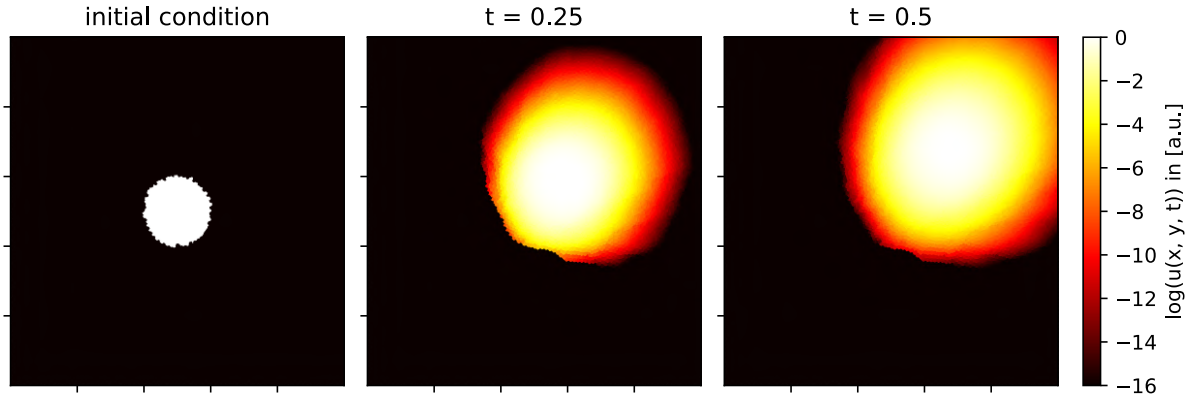

Figure 34: The advected circle, on a Voronoi mesh with $N = 10000$ cells, is slightly asymmetric. This can be seen in Logspace. (Video)

3.2.2 Shallow water equations

For the shallow water equations, a first-order Godunov scheme and a second-order MUSCL scheme, each with an HLL solver, are implemented. We perform different simulations as shown in Tab. 3.

Dim.	Initial cond.	N	Mesh	Order	slope-limiting	Comment
2D	horizontal dam break	10000	Cartesian, Voronoi	1st	-	verification 2D
2D	diagonal dam break	10000	Cartesian, Voronoi	1st	-	verification 2D
1D	gaussian	1000	Cartesian	1st	-	another solver
1D	dam break	100	Cartesian	1st, 2nd	on	higher-order
1D	dam break	100	Cartesian	2nd	off, on	slope-limting
2D	three Gaussians	62500	Voronoi	1st, 2nd	on	higher-order 2D
2D	dam break onto box	62500	Cartesian	2nd	on	
2D	gaussian onto circle	44250	Voronoi	2nd	on	
2D	dam break onto coastline	38690	Voronoi	2nd	on	

Table 3: Shallow water simulations

Dam Break

We start by testing the first-order scheme using the dam break problem, which is the Riemann problem without initial velocity for SWE. Since an analytical solution to the dam break is known (Delestre et al., 2013), we use it to verify the correctness of our solvers in 2D. In Fig. 35, the analytical solution is shown together with the simulation results for a horizontal and diagonal dam break on Cartesian and Voronoi meshes.

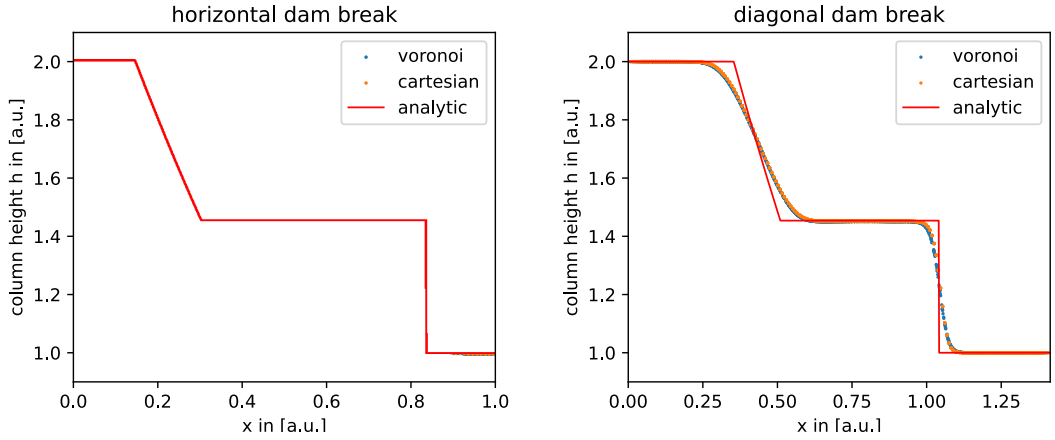

Figure 35: Simulation results for a dam break with $N = 10000$ compared to analytical solution. Left: horizontal dam break. Right: diagonal dam break. Both times, the solution looks like the analytical solution plus diffusion.

As one can see, the dam breaks are simulated correctly by the scheme (apart from diffusion, of course). The randomness of the Voronoi mesh adds a small noise to the solution, while the Cartesian solution stays effectively 1D. For the diagonal dam break, we only look at the states close to the diagonal line $x = y$. This is because the zero gradient

boundary condition still has tiny reflections (see Fig. 36). This is a boundary effect, which should be excluded from the dam break test scenario.

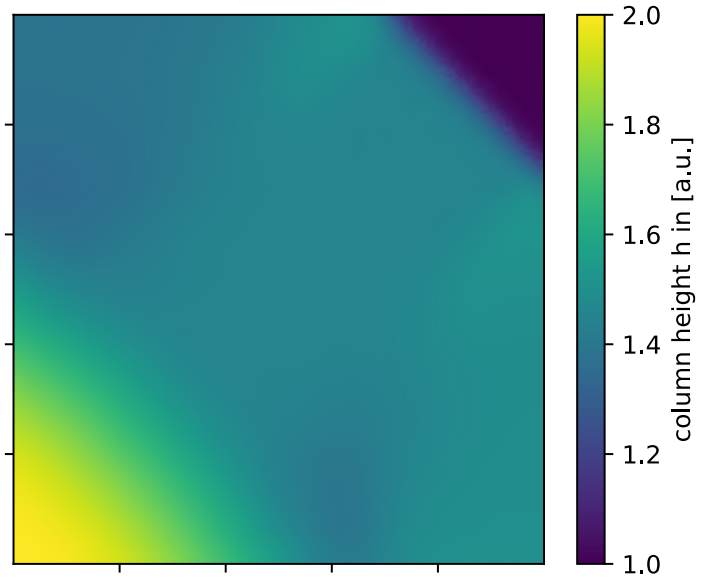

Figure 36: Zero gradient boundary condition leading to small reflections. We exclude this for the diagonal dam break test.

Smooth solutions

Before going to higher-order, we want to compare our simulation to a smooth solution. For that, we use the 1D Python SWE_Solver by Daniel Cortild, which can be installed using the pip package manager (Cortild, 2023). By setting the same smooth initial conditions, we can then compare our solver to another working solver. This could be done for arbitrary initial conditions. For a Gaussian initial condition, the evolution of both solvers is shown in Fig. 37. Both solutions show the same behavior. However, for the same resolution, the Godunov scheme is a bit less diffusive than the Python solver.

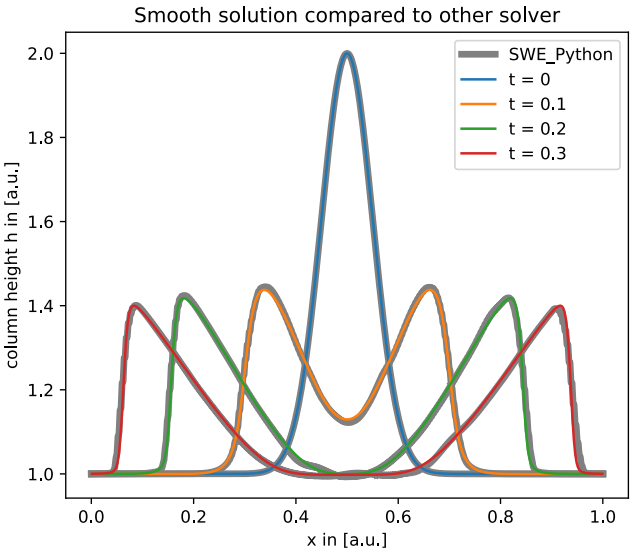

Figure 37: Initial Gaussian solved with a Python SWE solver and the first-order Godunov scheme on a 1D Cartesian mesh. (Video)

Second-order

Since we now have a working first-order solver, we will test the second-order scheme next. We start by comparing the solution to a dam break again. The importance of using slope limiters can be seen in Fig. 38. Without slope limiting, the solution oscillates near discontinuities. We, therefore, generally use slope limiting when using the MUSCL scheme. At the same resolution, the slope limited MUSCL scheme looks way more accurate and seems to have less diffusion.

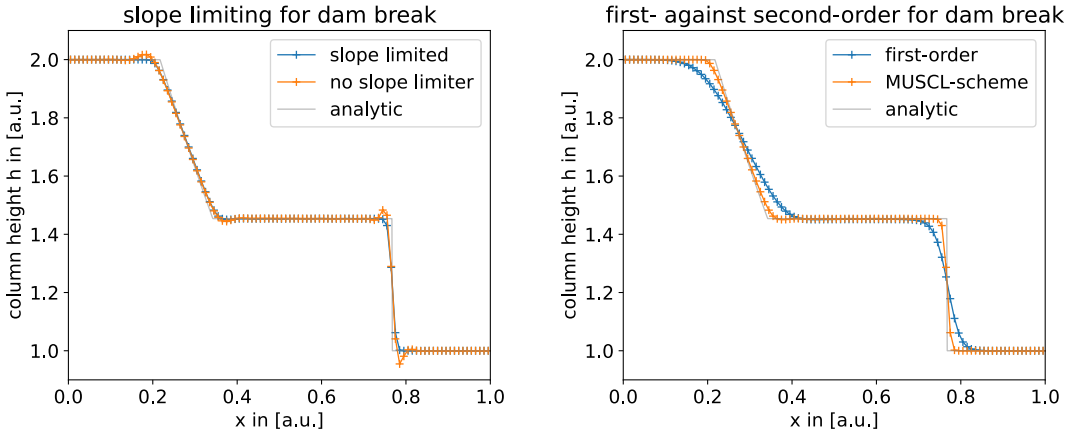

Figure 38: Dam break with $N = 100$. Left: MUSCL scheme with and without slope limiting. Right: MUSCL scheme against first-order for the same resolution.

This is also visible in 2D when simulating three initial Gaussians with periodic boundary conditions (see Fig. 39). It almost looks like there is some kind of blur on the first-order simulation. Clearly, the shock resolution is way better in a second-order scheme.

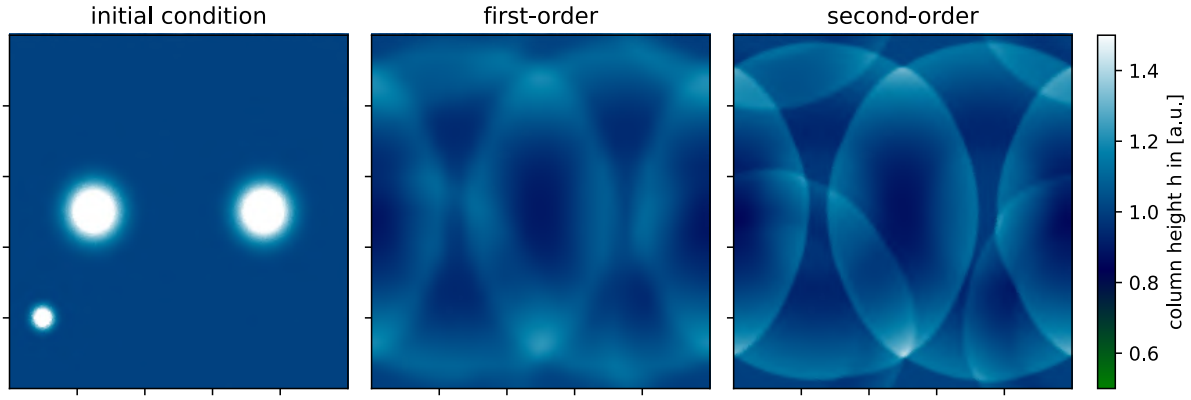

Figure 39: Three initial Gaussians with periodic boundary conditions. For a Voronoi mesh with $N = 62500$ cells, the evolution at $t = 0.35$ is shown for the first- and second-order schemes.

We also want to verify the better accuracy using the L1 error estimate. In Fig. 40, the L1 error over the resolution for different simulations and schemes is shown. We compare the first-order Godunov scheme to the second-order MUSCL scheme. We do this for the dam break, which is a heavily slope limited simulation, and for a very smooth Gaussian simulation. Since we do not have an analytic solution for the evolution of such smooth

simulations, we use a very high resolution first-order run as an approximate analytic solution. This way we can get the convergence for smooth regions as well. However, one has to be careful to stay way above the error of that high resolution run with any other run. Otherwise, the error will just converge to a constant value, which gives us no information on scaling. Interestingly, it makes a huge difference whether we look at a discontinuous or smooth solution. We can see that the slope limiting of the dam break simulation limits the convergence to first-order for the MUSCL scheme, which scales way better for the smooth solution than first-order.

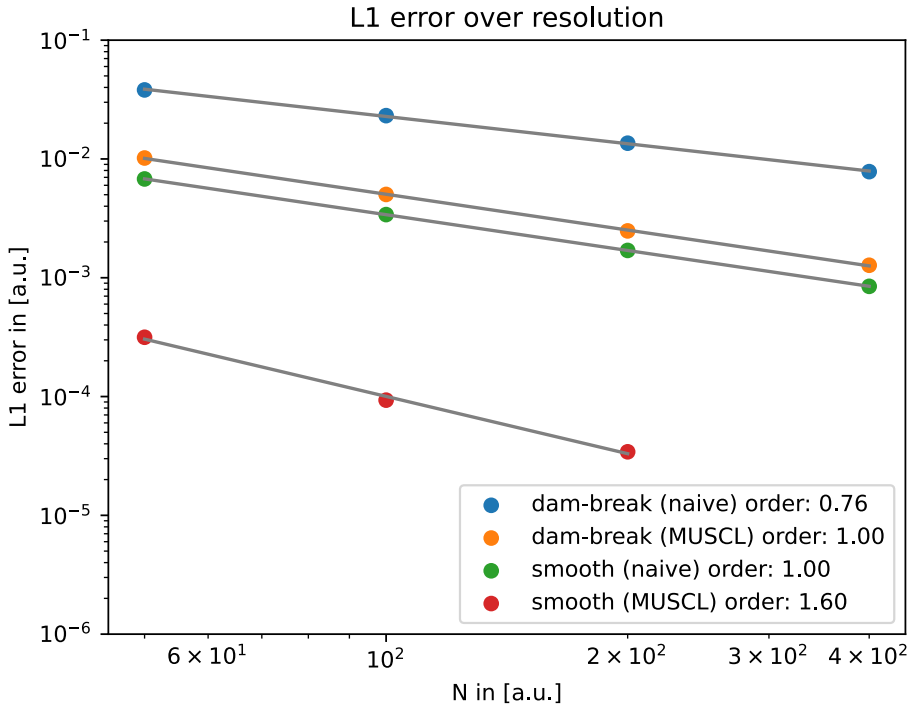

Figure 40: L1 error over N for first-/second- order and dam break or smooth solution.

However, it is not entirely clear why the scaling is not exactly second-order but lower. The first-order scheme also only archives first-order convergence for the smooth solution. Generally, one can conclude that the overall error of the second-order scheme in smooth regions is more than an entire magnitude smaller than the error of the first-order scheme for the range of resolution we can simulate in a reasonable time. This more than justifies extending a scheme to second-order.

Custom inner boundaries

We now want to use the second-order solver to model the waves around custom inner boundaries to demonstrate usecases of the solver. The boundary can be seen in green, while the shallow water is blue to white, depending on height. The first initial condition is a dam break on a rectangular box. One can see in Fig. 41 that the dam break reflects on the front side of the box while bending around the box on the backside. After that, the reflection itself gets reflected by the outer boundary, and the undisturbed dam break reaches the end of the box and reflects as well. After the reflection, the two bending waves behind the box meet and overlap.

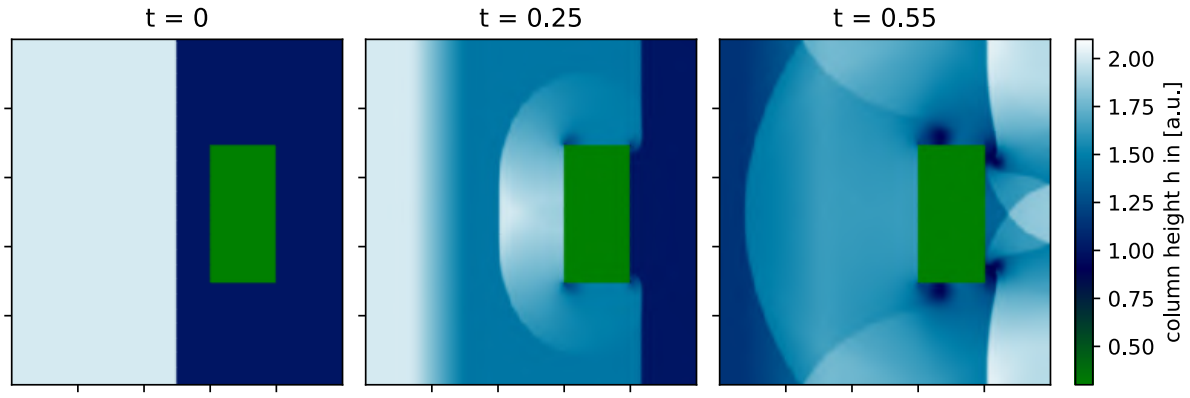

Figure 41: An initial dam break hitting a custom boundary box using a 250×250 Cartesian mesh and reflective boundary conditions.

Next, we look at a custom-shaped circular boundary. In Fig. 42, the initial Gaussian wave expands around the circle and is reflected by the outer boundary. Afterward, the original and reflected waves reflect again from the outer and inner boundaries.

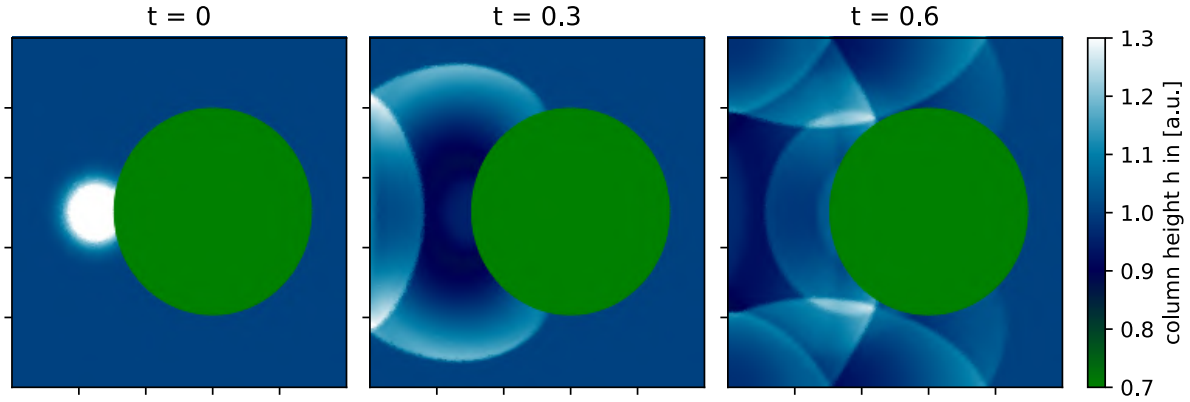

Figure 42: An initial Gaussian hitting a custom boundary circle on a Voronoi mesh with $N = 44250$ cells and reflective boundary conditions.

Finally, we want to look at a custom boundary on a Voronoi mesh, shaped roughly like Europe and Africa, focussing on the Strait of Gibraltar (see Fig. 43). This is by no means an accurate representation of the continents, but it is a nice example of shallow water waves along a coastline with a channel of decreasing width. In addition to the green boundaries and the blue water, the flow velocity is represented with small grey arrows. For an initial dam break shifted to the left, we can see that the wave hits Africa first and is reflected at the bottom. Later, the wave travels through the narrowest part (i.e. the Strait of Gibraltar) and reaches maximum height, with many reflections. One can see that the flow converges into the channel while expanding afterward. Additionally, a wave reflects from Portugal, and water flows into the Biskaya. In the third frame, one can see the flow getting broader and the wave getting smaller in the Mediterranean Sea. Instead, the flow into the Biskaya has stopped, and the water is high there.

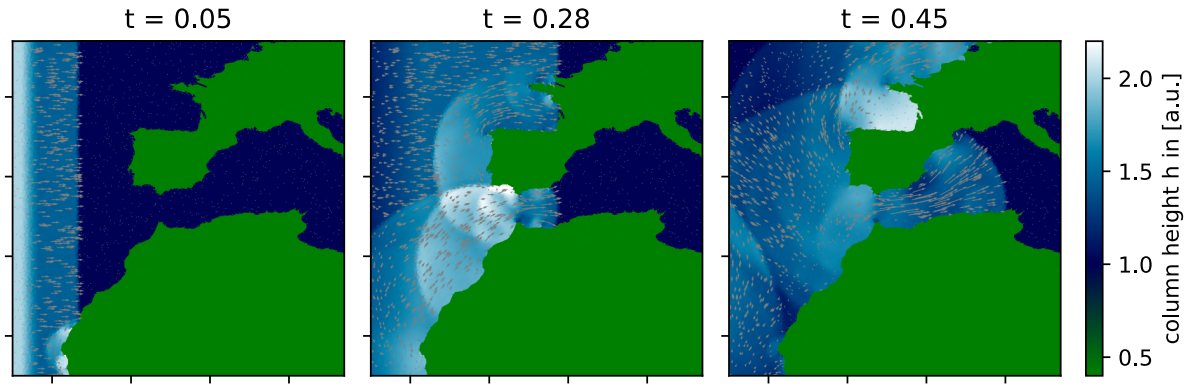

Figure 43: An initial dam break hitting a custom boundary shaped like the Strait of Gibraltar, using a Voronoi mesh with $N = 38690$ cells. Coastline data: naturalearthdata (Video)

These examples show that this solver, in principle, could be used to model various physical problems. In addition to the flow around boundaries on a constant ocean surface, one could add an uneven ocean surface by adding a force term. We will do something similar for the Rayleigh Taylor instability for the Euler equations later. With an uneven ocean surface, one could also add dry/wet regions such that the column water height could become zero in some places. For that, a more careful numerical handling would be needed, since we divide by the column water height in the current update scheme. Additionally, one could add further force terms like Coriolis forces.

3.2.3 Euler equations

Finally, we want to look at implementing Euler equations using the finite volume method. Implemented are again a first-order Godunov scheme and a second-order MUSCL-Hancock scheme, with an HLL Riemann solver each. Since we used the general state vector notation in deriving the scheme, adapting the scheme from shallow water to Euler is straightforward. One only needs to replace the state vectors in the code. Tab. 4 shows the simulations we ran.

Dim.	Initial cond.	N	Mesh	Order	slope-limiting	Comment
1D	shock tube	100	Cartesian	1st	-	verification
1D	shock tube	100	Cartesian	2nd	non TVD	verification
1D	shock tube	100	Cartesian	2nd	off, non TVD, TVD	diff. slope-limiting
2D	horizontal shock tube	40000	Cartesian, Voronoi	1st, 2nd	non TVD	verification 2D
2D	diagonal shock tube	40000	Cartesian, Voronoi	1st, 2nd	non TVD	verification 2D
2D	Kelvin Helmholtz	22500	Voronoi	2nd	non TVD	low-res
		160000	Voronoi	2nd	non TVD	high-res
		10000, 16000	adapted Voronoi	2nd	non TVD	adapted mesh
2D	Rayleigh Taylor	25200	Voronoi	2nd	non TVD	low-res
		73000	Voronoi	2nd	non TVD	high-res
		10000	Voronoi	2nd	non TVD	diff. Atwood nr.
2D	2D Riemann	62500	Voronoi	2nd	non TVD + TVD	low-res
		250000	Voronoi	2nd	non TVD + TVD	high-res
2D	Vortex shedding	49400	adapted Voronoi	2nd	non TVD	
2D	Airfoil	22000	Voronoi	2nd	non TVD	$\alpha = 0°, 10°, 25°$

Table 4: Euler equation simulations

Shock tube

We again use the Riemann problem with zero initial velocity (shock tube, see Fig. 22) to verify the implementation of our code. Using the first-order scheme, we look at the 1D case for density, velocity, total energy per unit volume and pressure. As one can see in Fig. 44, the solver simulates the structure of the shock tube correctly apart from numerical diffusion. Interestingly, the contact discontinuity (the middle one in the density) is the most diffusive. This is expected because we used the HLL solver.

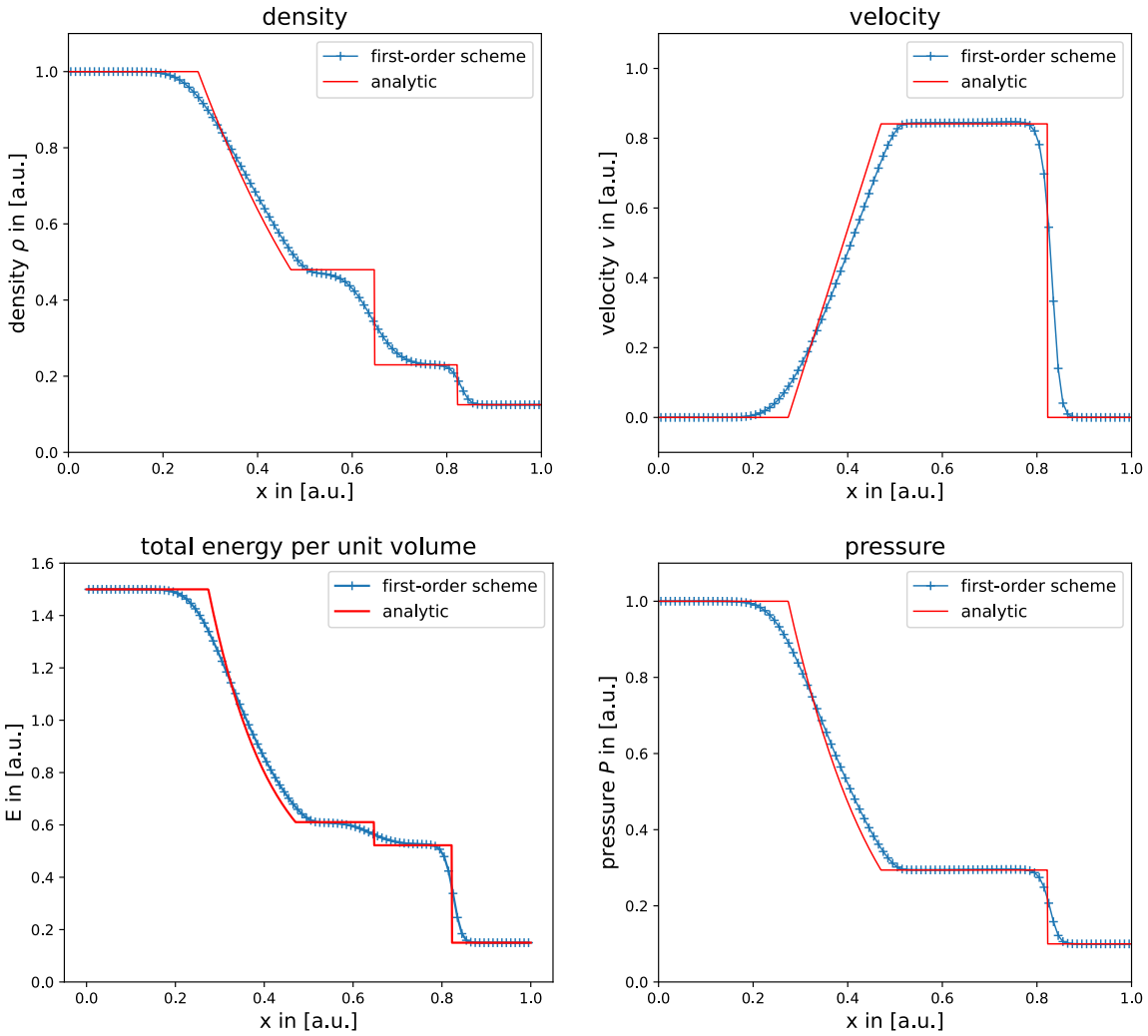

Figure 44: 1D shock tube test for first-order Godunov scheme with $N = 100$ on a Cartesian mesh and zero gradient boundaries.

If we compare the first-order solution to the second-order scheme, we see that the error is much smaller for the second-order scheme (see. Fig. 45). The contact discontinuity here, again, is by far the most diffusive. In addition to that, the non total variation diminishing (TVD) slope limiter (Springel, 2010), we use, still leads to small post-shock oscillations. This, however, is not problematic here since they do not grow over time and shrink with resolution. For simulations with strong shocks, this, however, could become problematic. There, one might change to the TVD slope limiter we introduced (Duffell and MacFadyen, 2011). For the density, one can see in Fig. 46 the two different slope limiters used, compared to a not limited solution. As one can see, the solution without slope limiting shows clear oscillations, while the non TVD slope limited solution has much smaller oscillations. The TVD slope limiter for $\theta = 1$ also shows small oscillations, while no oscillations occur for $\theta = 0.5$.

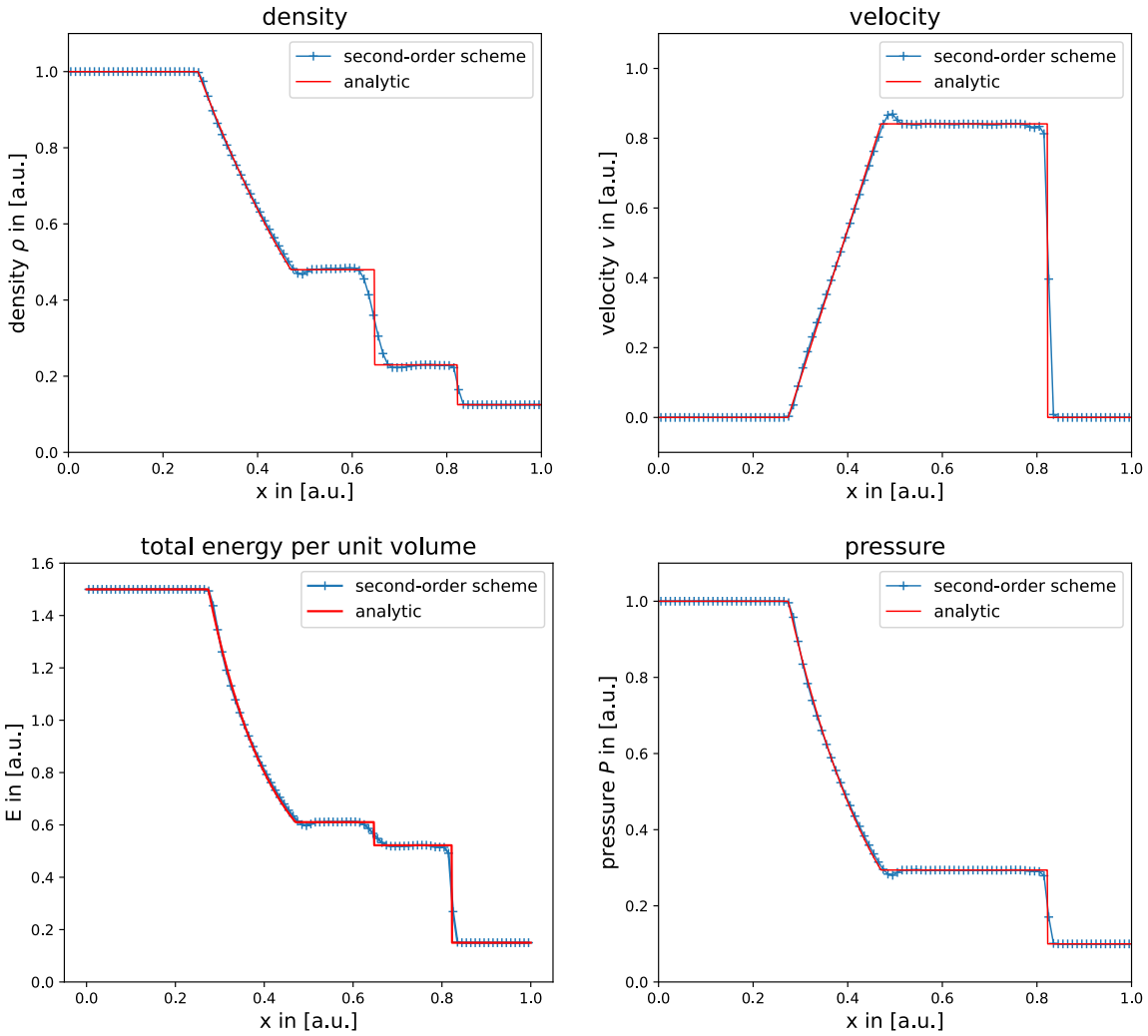

Figure 45: 1D shock tube test for second-order MUSCL-Hancock scheme with $N = 100$ on a Cartesian mesh with zero gradient boundaries and non TVD slope limiter.

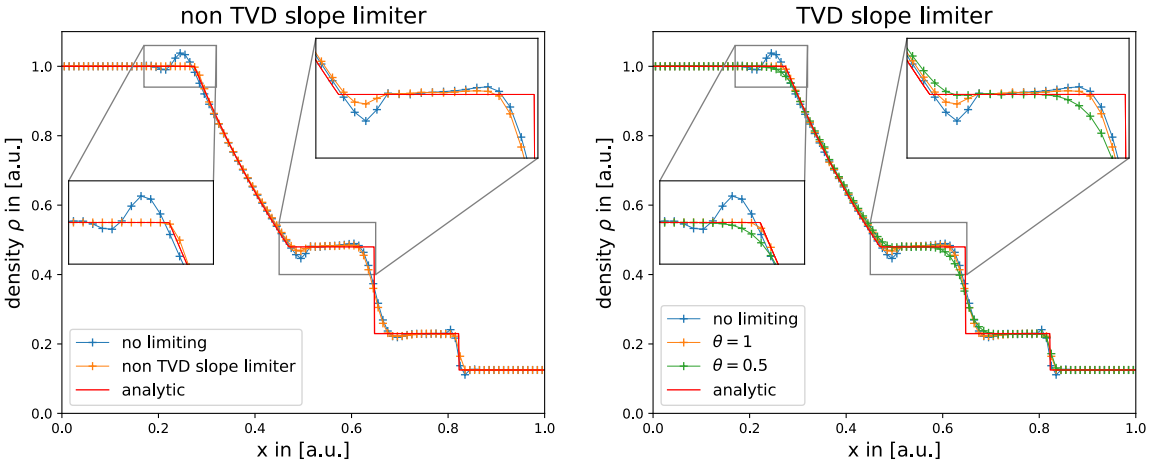

Figure 46: 1D shock tube test for $N = 100$ on a Cartesian mesh with zero gradient boundaries. Non TVD and TVD slope limiter compared to not limited solution.

We still have to verify the scheme in 2D. In Fig. 47, one can see a horizontal shock tube on the left, while on the right, one can see a diagonal shock tube. Both capture the structure of the shock tube correctly on 2D Cartesian and Voronoi meshes for the first-order algorithm. Similar tests have been done for the second-order as well.

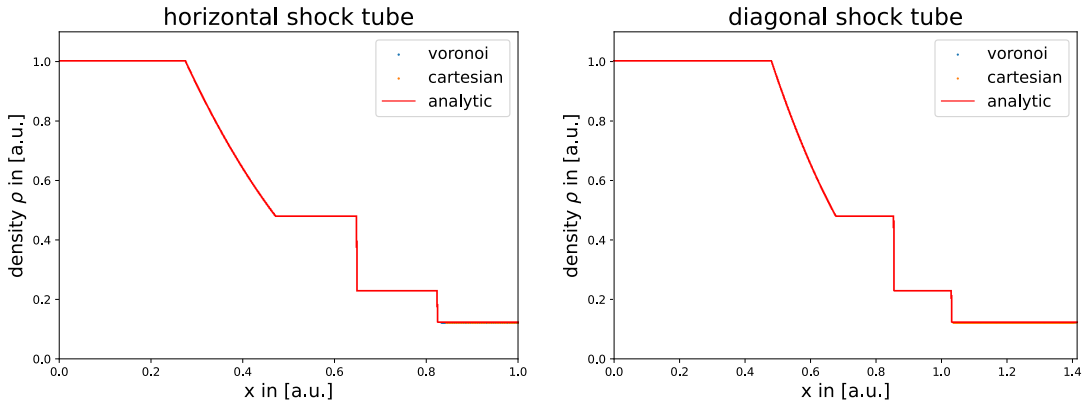

Figure 47: Horizontal and diagonal 2D shock tube tests on Voronoi and Cartesian using $N = 40000$ and first-order.

Again, we only plot the seeds close to the diagonal line for the diagonal shock tube to exclude the boundary effects of the reflections.

Since the shock tube test works as expected and the algorithm is conceptually the same as for shallow water, we will not focus on further verifying it with e.g. other codes for a smooth solution. This is left to do for future work. Instead, we will look at some of the classical hydrodynamic problems and see if the code reproduces them qualitatively. For that, we will only use the second-order scheme from now on.

Kelvin-Helmholtz instability

The Kelvin-Helmholtz instability is perhaps one of the most prominent test problems in hydrodynamics. It occurs when there is a velocity shear in a fluid or a velocity difference on a surface between two fluids.

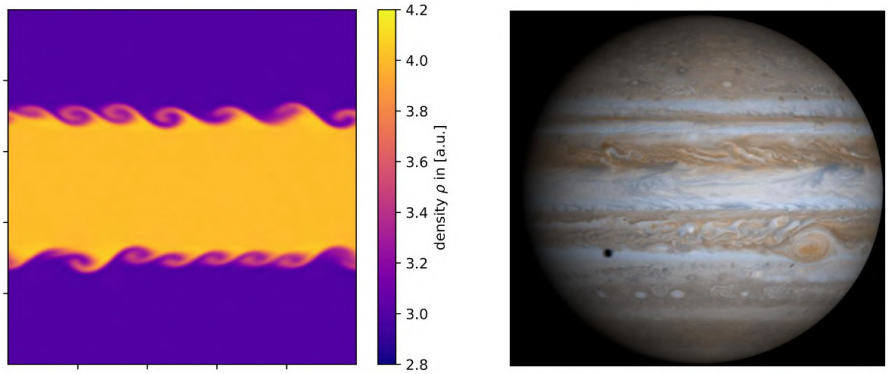

Figure 48: Left: Shear flow between two parts of the fluid inducing a Kelvin-Helmholtz instability simulated on a Voronoi mesh with $N = 160000$. The middle part flows to the right, and the upper and lower part flows to the left. Right: Kelvin-Helmholtz instabilities on Jupiter indicated by clouds captured by Cassini. (NASA/JPL/University of Arizona)

If the initial state gets perturbed slightly, eddies at the boundary surface start to occur and grow. They form, merge into larger eddies and transport macroscopic energy from the velocity difference down to the viscosity scale, where it then diffuses into heat. (Bartelmann, 2021) The smallest possible scale of eddies is a size comparable to the mean free path length of the particles (Bartelmann, 2021). We, therefore, expect to see the simulation resolution affecting the smallest scales of the eddies. Since there are many scenarios where a fluid has a velocity shear, the Kelvin-Helmholtz instability occurs widely in nature. In Fig. 48, a simulated Kelvin-Helmholtz instability is shown next to the clouds on Jupiter. Clearly, similar structures are visible.

Following (Bartelmann, 2021), we consider a system (see Fig. 49) of two fluids separated by a boundary. For $y > 0$, the fluid has the density ρ_1 and velocity v. The lower fluid ($y < 0$) instead has the density ρ_2 and no initial velocity. We describe the boundary between the two fluids as $\zeta(x,t)$ and perform an initial perturbation in the form of

$$\delta\vec{v} = \delta\vec{v}_0 e^{i(kx-\omega t)} \tag{3.46}$$

$$\delta P = \delta P_0 f(y) e^{i(kx-\omega t)} \tag{3.47}$$

$$\zeta = \zeta_0 e^{i(kx-\omega t)}. \tag{3.48}$$

In addition to that, we assume the fluid to be incompressible ($\vec{\nabla}\cdot\delta\vec{v} = 0$). Even though we are solving the compressible Euler equations, this is a good approximation for velocities smaller than 0.3 of the sound speed (Anderson, 2007).

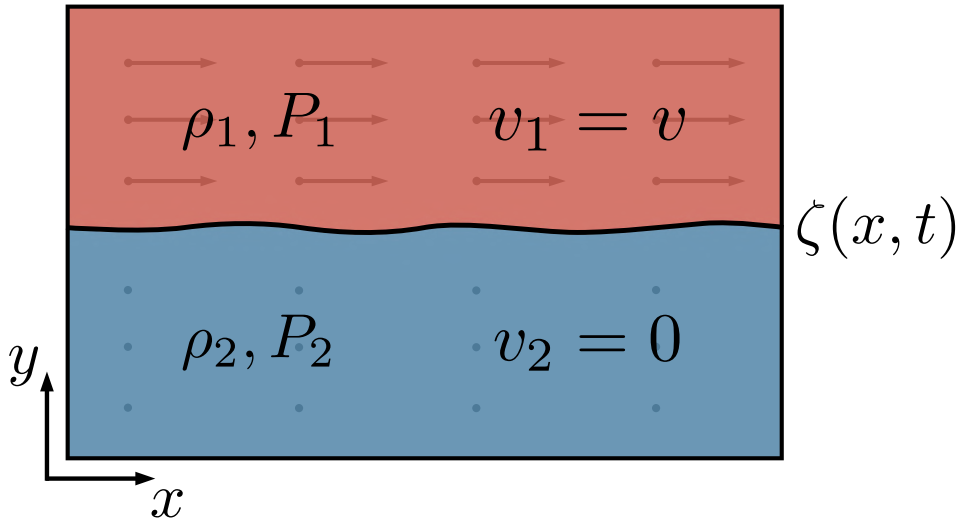

Figure 49: Initial conditions for a Kelvin-Helmholtz instability.

One can then derive a dispersion relation for the linearized Kelvin-Helmholtz instability (Bartelmann, 2021).

$$\boxed{\omega_\pm = \frac{kv}{\rho_1+\rho_2}\left(\rho_1 \pm i\sqrt{\rho_1\rho_2}\right)} \tag{3.49}$$

The frequency is imaginary if the velocity and both densities are different from zero. An imaginary frequency plugged into $\delta\vec{v}$, δP, ζ leads to an exponential growth, showing that this situation is unstable. The exponential growth, however, is just a result of the

linearization and, therefore, unphysically for later phases of the growth.

For the implementation results, we first want to verify that the lowest scale of the eddies indeed depends on the mesh resolution. For that, we can see a low-resolution simulation on the left and a higher-resolution version on the right. The initial condition here is the shear flow without any external perturbation. The only perturbation, therefore, is Voronoi mesh noise.

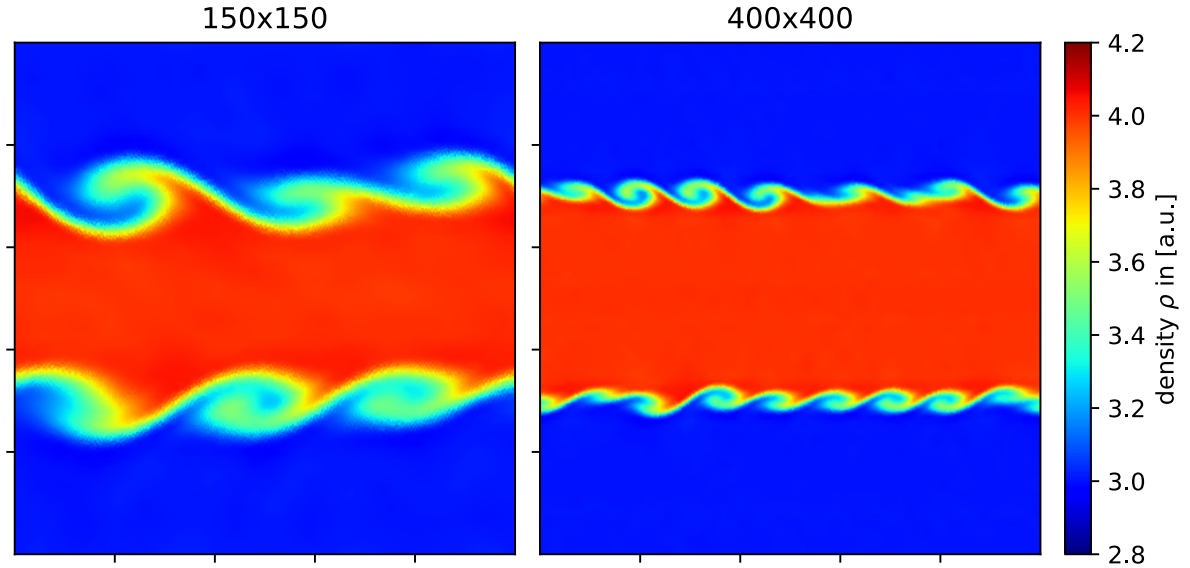

Figure 50: Left: Low resolution KH-instability $N = 22500$. Right: High-resolution KH-instability, $N = 160000$. We use periodic boundary conditions on a Voronoi mesh with perturbation only given by Voronoi mesh noise. For the middle part of the flow we set $\rho_1 = 4$, $\vec{v}_1 = (0.1, 0)$, $P_1 = 1$ and for the upper and lower part of the flow $\rho_2 = 3$, $\vec{v}_2 = (-0.1, 0)$, $P_2 = 1$.

The size of the first eddies appearing in the low-resolution simulation is larger than in the higher-resolution simulation (see Fig. 50). It looks like the smallest eddies need a minimum number of cells to form and, therefore, depend on the mesh resolution.

Next, we look at the time evolution of the high resolution simulation. As shown in Fig. 51, the initial mesh starts out nearly unperturbed. After two seconds, small eddies start to form, merging into larger eddies or diffusing out. Since the initial perturbation is only given by the Voronoi mesh noise, the upper and lower eddies occur at different positions and strengths. After more time has passed, the turbulent eddies become more irregular as the main flow dissolves more and more.

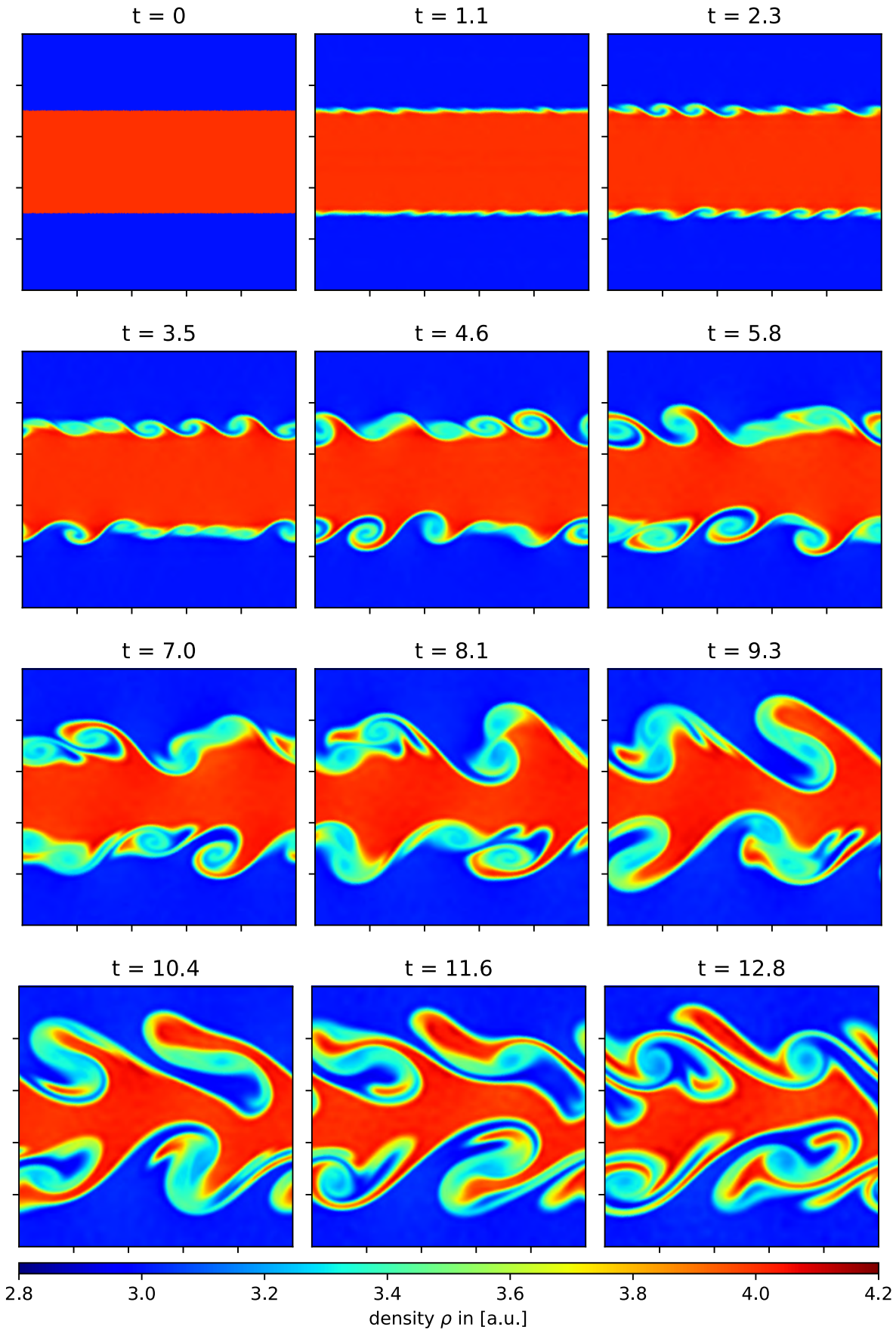

Figure 51: Time evolution of the Kelvin-Helmholtz instability on a Voronoi mesh with periodic boundary conditions and $N = 160000$. (Video)

As one can see, there are many places in the flow where the density is not changing much. To efficiently simulate this, it makes sense to increase resolution, where more is going on and to decrease resolution, where less is going on. A very naive way to do this is to increase the resolution near the initial boundary surfaces. In Fig. 52, this is done in the shape of a double Gaussian plus some mesh regularization. This simple approach works well as long as the eddies are small enough to stay inside the well-resolved area.

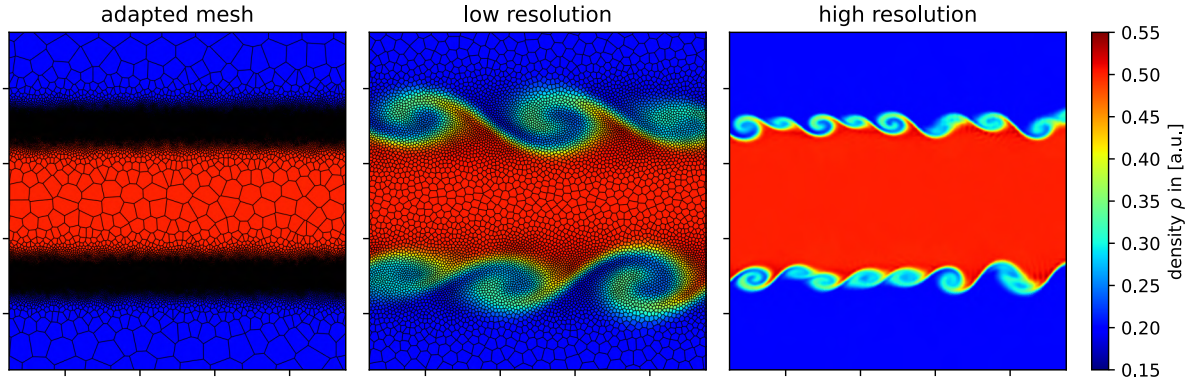

Figure 52: Kelvin-Helmholtz Instability. On the left, the grid is shown in black, plus the initial condition behind that. In the middle and right plots, low- and high-resolution versions show the increased resolution near the initial boundary surfaces. Mid: $N = 10000$. Right: $N = 160000$.

A more advanced way to do this is to adaptively refine and derefine the mesh by splitting a cell in two or combining two cells. One could, for example, increase resolution based on the density gradient in the cells. This is left for future work but has been widely done in, e.g., (Springel, 2010).

Rayleigh-Taylor instability

The next hydrodynamic test problem we will look at is the Rayleigh-Taylor instability.

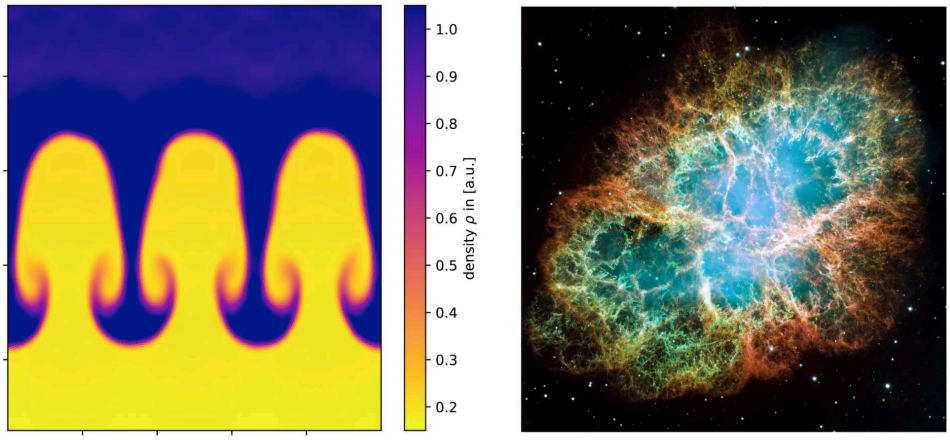

Figure 53: Left: A heavier layer of fluid above a lighter layer inducing a Rayleigh-Taylor instability on a Voronoi mesh with $N = 62500$. Right: Rayleigh-Taylor instabilities in the filamentary structure of the supernova remnant called Crab Nebula (NASA, ESA).

It occurs when a lighter fluid layer pushes against a heavier fluid layer. One then gets growing mushroom-like structures, which additionally become Kelvin-Helmholtz unstable (see Fig. 53). This, for example, can happen when a heavier fluid layer lies above the lighter fluid under the influence of gravity. In astrophysics, one can observe the Rayleigh-Taylor instability, for example, in supernova remnants. There, less dense, hot gas from the inside is pushing against colder, denser gas further out (see Crab Nebula in Fig. 53).

We again start looking at the stability following (Bartelmann, 2021). For that, we consider two different fluids meeting at an unperturbed surface ($y = 0$), perpendicular to the gravitational acceleration g (see Fig. 54). The upper fluid has density ρ_1 and height h_1, while the lower fluid has density ρ_2 and depth h_2. Both fluids have no initial velocity. The initial pressure is given by the hydrostatic equilibrium:

$$P_1 = P_0 + \rho_1 g y \quad \text{for } y > 0 \tag{3.50}$$

$$P_2 = P_0 + \rho_2 g y \quad \text{for } y < 0. \tag{3.51}$$

If the surface is now perturbed, described by a function $\zeta(x, t)$, we want to know under which condition an instability occurs.

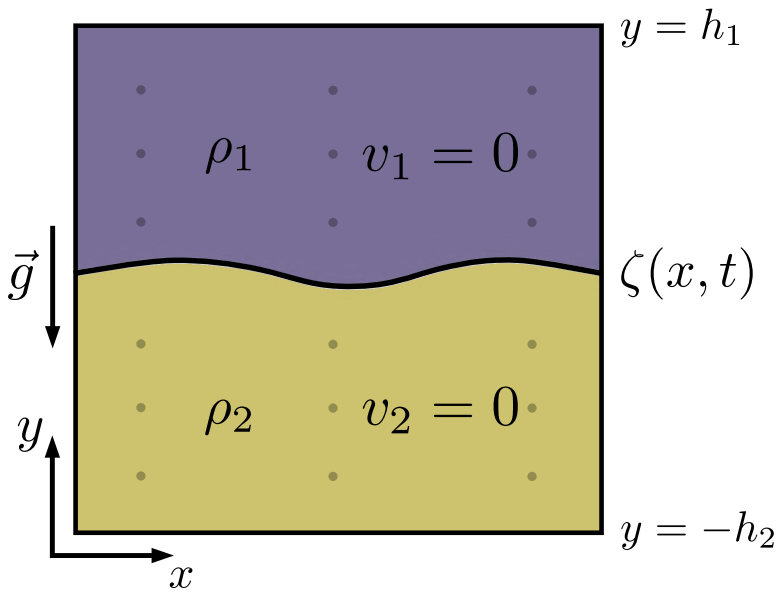

Figure 54: Initial conditions for a Rayleigh-Taylor instability.

A linear stability analysis following similar assumptions as for the Kelvin-Helmholtz instability then leads to the following dispersion relation (Bartelmann, 2021):

$$\boxed{\omega^2 = \frac{kg(\rho_2 - \rho_1)}{\rho_2 \coth(kh_2) + \rho_1 \coth(kh_1)}} \tag{3.52}$$

It shows the interesting result, that the frequency ω becomes imaginary if $\rho_1 > \rho_2$, indicating, that there is an instability, if the heavier fluid is above the lighter fluid.

Include gravity in FV update scheme

Before looking at the instability, we have to adapt the finite volume update scheme to include the constant gravity. For that, we add an external force term $\mathbf{S}_i$ to the right side of the general hyperbolic PDE, i.e. Eq. (1.7).

$$\frac{\partial \mathbf{U}}{\partial t} + \vec{\nabla} \cdot \vec{\mathbf{F}} = \mathbf{S} \tag{3.53}$$

The force term for constant gravity is given by

$$\mathbf{S}_i = \begin{pmatrix} 0 \\ \rho_i g_x \\ \rho_i g_y \\ \rho(u_i g_x + v_i g_y) \end{pmatrix} \tag{3.54}$$

Following the derivation of the finite volume scheme completely analogous to before leads to an adapted version of Eq. (3.9).

$$\boxed{\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \frac{\Delta t}{A} \sum_{j \in \partial C_i} \vec{\mathbf{F}}_j^{\star}(\mathbf{U}_L^n, \mathbf{U}_R^n) \cdot \vec{n_j} \cdot l_j + \Delta t \mathbf{S}_i^n} \tag{3.55}$$

In addition to that, we set the top and bottom boundaries to reflective while the left and right boundaries are periodic to ensure the correct initial conditions.

We start by looking at the time evolution of the simulation on a Voronoi mesh with in total $N = 62500$ cells (see Fig. 55). In the frames, a section containing $N \approx 25200$ cells is shown. The sinusoidal perturbation starts to grow somewhat linearly first before later becoming Kelvin-Helmholtz unstable on its sides. The resolution here is so low that the lowest scale eddies are roughly on the scale of half the Rayleigh-Taylor instability. We compare this to a higher resolution simulation with $N = 180625$ cells, where again, a subsection of the simulation box is shown containing $N \approx 73000$ cells (see Fig. 56). Here, the tiniest eddies are smaller and appear more clearly. This is consistent with our findings for the Kelvin-Helmholtz instability.

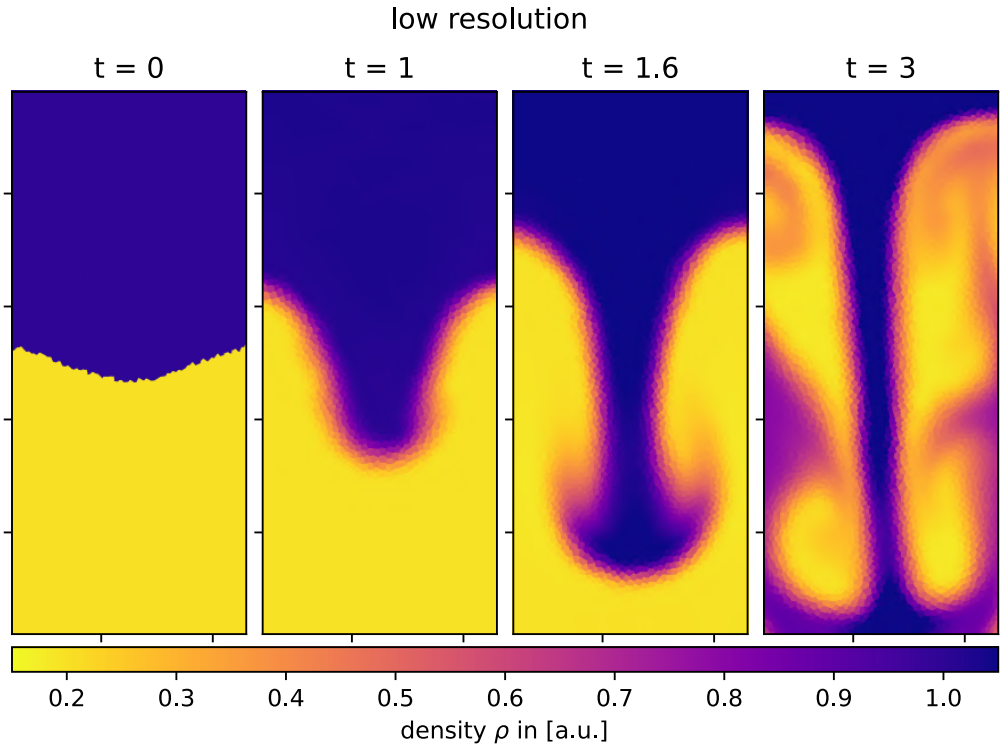

Figure 55: Time evolution of Rayleigh-Taylor instability for $N \approx 25200$ Voronoi cells. We set $\rho_1 = 1$, $\rho_2 = 0.2$, the pressure at the boundary layer to $P = 1$ and use $\vec{g} = (0, -1)$.

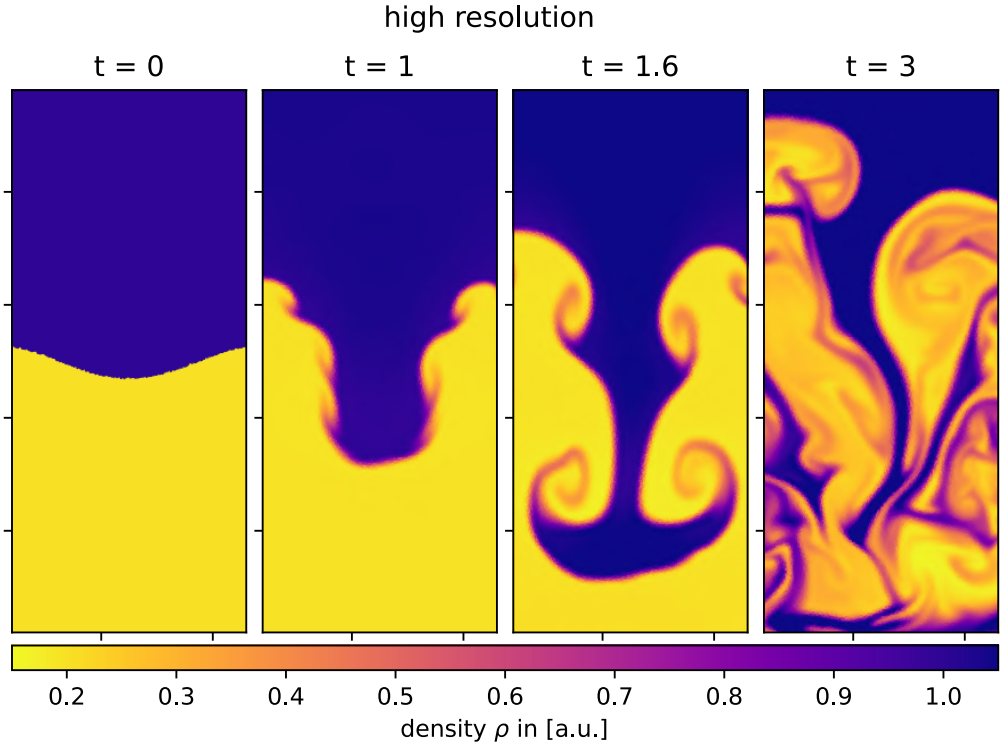

Figure 56: Time evolution of Rayleigh-Taylor instability for $N \approx 73000$ Voronoi cells. We set $\rho_1 = 1$, $\rho_2 = 0.2$, the pressure at the boundary layer to $P = 1$ and use $\vec{g} = (0, -1)$. (Video)

The second thing we want to consider, apart from the resolution, is the effect of the density contrast. The dimensionless Atwood number used to characterize this is a density ratio given by

$$A = \frac{(\rho_1 - \rho_2)}{\rho_1 + \rho_2}. \tag{3.56}$$

We look at the shape of the Rayleigh Taylor instability at $t = 1.5$ for different Atwood numbers.

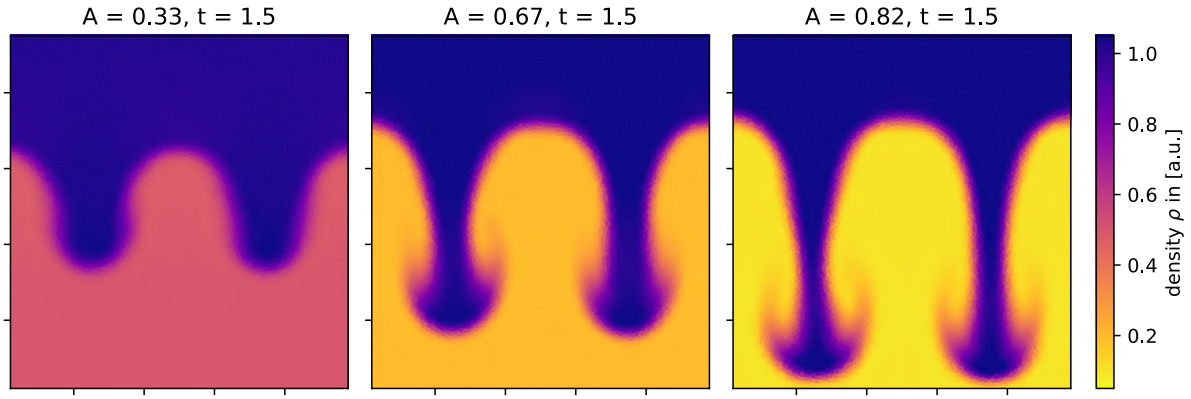

Figure 57: Time evolution of Rayleigh-Taylor instability for different Atwood numbers and $N = 10000$ Voronoi cells.

Fig. 57 shows that the growth of the instability is faster for higher density contrasts, i.e. higher Atwood numbers.

2D Riemann problem

Next, we want to simulate a 2D Riemann problem proposed by (Kurganov and Tadmor, 2002). The initial condition for that is divided into four quadrants and given in Tab. 5.

$p_2 = 0.3$	$\rho_2 = 0.5323$	$p_1 = 1.5$	$\rho_1 = 1.5$
$u_2 = 1.206$	$v_2 = 0$	$u_1 = 0$	$v_1 = 0$
$p_3 = 0.029$	$\rho_3 = 0.138$	$p_4 = 0.3$	$\rho_4 = 0.5323$
$u_3 = 1.206$	$v_3 = 1.206$	$u_4 = 0$	$v_4 = 1.206$

Table 5: Initial conditions to the 2D Riemann problem

In Fig. 58, we compare a result of (Kurganov and Tadmor, 2002) to a second-order simulation on a Voronoi mesh with reflective boundary conditions.

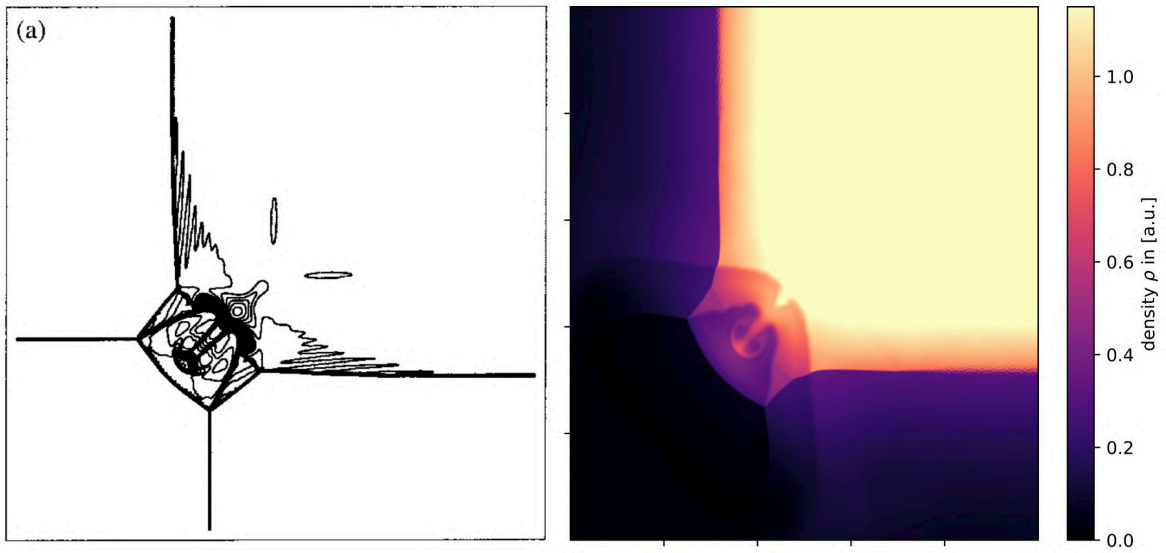

Figure 58: 2D Riemann problem. Left: Figure out of (Kurganov and Tadmor, 2002). Right: Simulation on a Voronoi mesh with $N = 250000$ and reflective boundary conditions. The mesh is adapted to have higher resolution in the bottom left quarter.

One can see the same structure developing, although the turbulence in this structure is highly dependent on the resolution. The time evolution can be seen in Fig. 59.

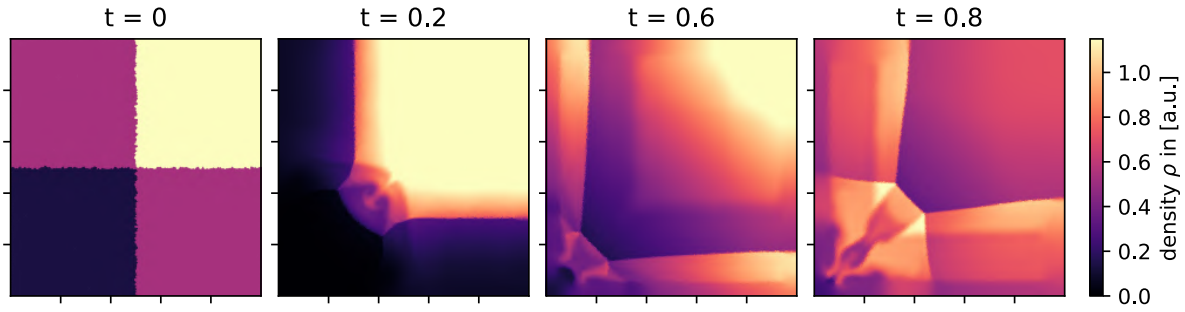

Figure 59: Time evolution of 2D Riemann problem on a Voronoi mesh with $N = 62500$. (Video)

Vortex shedding

When a fluid flows around a circular boundary for low viscosity, the flow becomes turbulent. Behind the circle, vortices occur and separate from the circle, leading to a vortex street. In Fig. 60, we also observe this phenomenon for $v = 0.3$ in the simulation. For the simulation, we choose a constant inflow boundary on the left boundary, just like in a wind tunnel. The other boundaries are set to zero gradient boundary conditions.

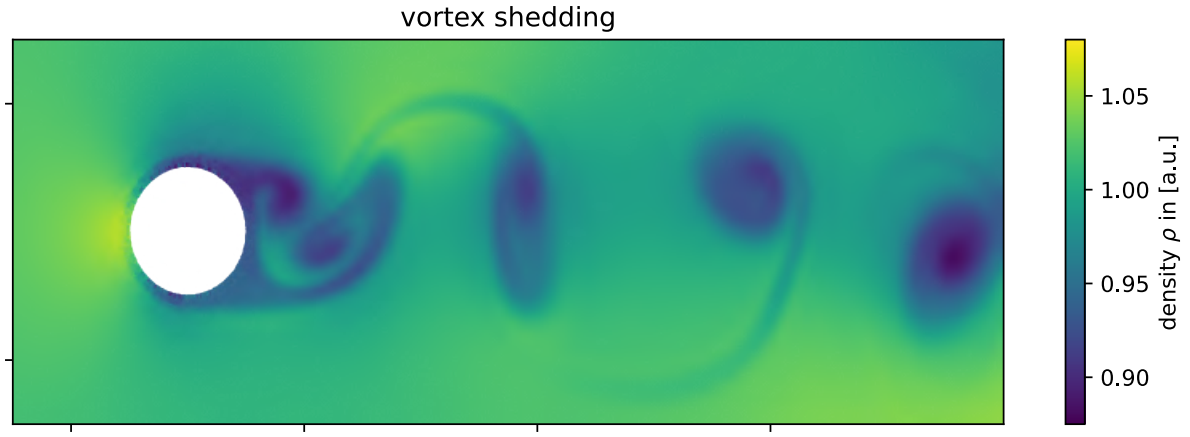

Figure 60: Vortex shedding around a circular boundary using $N \approx 49400$ Voronoi cells, zero gradient boundary conditions, and a wind tunnel-like inflow condition on the left. (Video)

Flow around an airfoil

The final internal boundary we want to simulate is an airfoil. The airfoil structure NACA 2412 is taken from the AeroPy package. The color indicates pressure, while the arrows indicate the flow direction. Fig. 61 shows that the pressure below the airfoil is generally higher than above, indicating that the wings generate lift. The lift could be calculated by vector-wise adding up the forces acting on all internal boundary faces. This calculation, however, will not be conducted in this thesis. For small angles of attack, like 0° or 10°, we observe laminar flow, while for higher angles of attack, like 25°, the flow separates from the upper part of the wing and turbulence occurs.

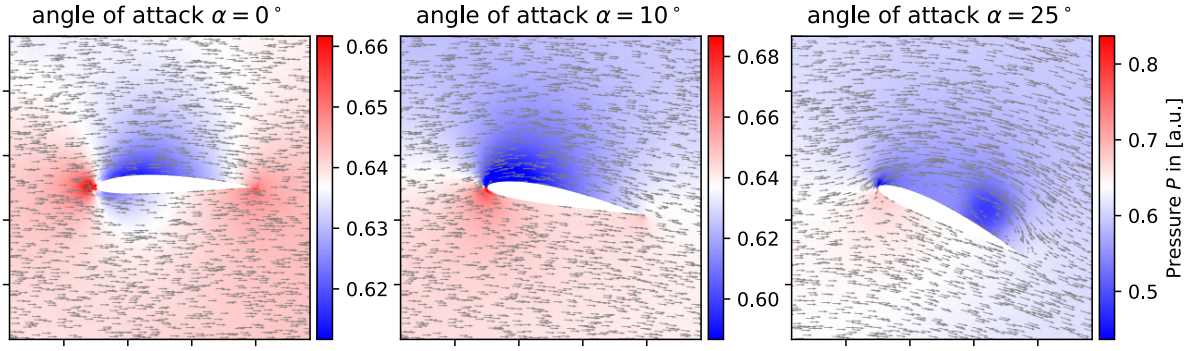

Figure 61: Flow around the NACA 2412 airfoil with different angles of attack on a Voronoi mesh using $N \approx 22000$ cells and zero gradient boundaries as well as a wind tunnel-like inflow condition on the left. White corresponds to normal pressure, while blue is lower and red higher pressure.

Recap: Finite Volume Methods

We introduced the finite volume method on unstructured grids, implementing upwind advection and first- and second-order Godunov-type schemes for the shallow water equations and Euler equations. We discussed an HLL Riemann solver, stability, boundary conditions, and slope limiting. After that, we observed numerical diffusion for the advection equation and showed that the finite volume scheme is conservative at machine precision. Additionally, we showed the Cartesian mesh introducing an angular dependence into the diffusive error, reducing it along the horizontal and vertical axis. After verifying the first- and second-order schemes for the shallow water equations, we observed that slope limiting around discontinuities decreases the convergence to below first-order. At the same time, on smooth solutions, the code can achieve higher-order convergence. We also demonstrated the potential of the code by looking at waves around various internal boundaries. For Euler equations, after verifying the scheme, we looked at different hydrodynamic problems like the Kelvin-Helmholtz or Rayleigh-Taylor instabilities, 2D Riemann problems, vortex shedding and the flow around an airfoil. The scheme could reproduce all the expected phenomena at a qualitative level. Small-scale features, like eddies forming, were limited by the mesh resolution, which is also consistent with our expectations. A more rigorous quantitative analysis of the performance of the code at these test problems is left for future work.

4 Discontinous Galerkin Methods

The discontinuous Galerkin scheme (DG) is a finite element method. Therefore, cells, in the finite volume sense, are called elements instead, and the physical quantities will be represented using functions on those elements. For DG, these functions are allowed to be discontinuous at element boundaries. The function on an element will be represented using basis functions. By choosing more advanced basis functions, a systematic way to achieve arbitrary higher-order spatial convergence exists while keeping the stencil small. To update an element, one only needs the immediate neighbors. The systematic way to achieve higher order makes DG very interesting to study and for applications (Knezevic, 2012). We will start by deriving the 1D DG scheme for scalar upwind advection and then look at its implementation using monomial basis functions. We will then generalize the method to 2D scalar upwind advection on a cartesian mesh (Schaal, 2016) and test the convergence of the implementation as well.

4.1 Towards a 1D scheme for advection

4.1.1 Basic formulation

We start with a hyperbolic PDE for a scalar quantity $q(x,t)$ on a 1D interval $\Omega \equiv (a,b)$.

$$\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} = 0 \tag{4.1}$$

Instead of this strong form of the equation, we will consider the weak form

$$\int_\Omega \left(\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} \right) v \; dx, = 0 \tag{4.2}$$

multiplying with an arbitrary test function v and requiring, that the integral over the domain Ω vanishes for any v (Knezevic, 2012).
Next, we will discretize space but keep time continuous for now. We divide Ω into $\{T_k\}_{k=1,\dots,K}$ elements. On each element, we will represent the solution as a polynomial of order N_p, which can be discontinuous from element to element. We represent the polynomial on T_k using a set of basis functions $\{\phi_j^k\}_{j=1,\dots,N_{bf}}$, that vanish outside of T_k and are constant in time. The numerical solution representation of q then is

$$q_h(x,t)\Big|_{T_k} = \sum_{j=1}^{N_bf} Q_j^k(t)\phi_j^k(x), \tag{4.3}$$

where $Q_j^k(t)$ are the solution coefficients at time t (Knezevic, 2012). Fig. 62 shows an example of the solution representation in DG. We later want to obtain an evolution equation for those coefficients. Representing q on T_k using the basis functions allows for any q_h out of the local function space

$$V_h^k \equiv \text{span} < \{\phi_j^k\}_{j=1,\dots,N_{bf}} > \tag{4.4}$$

and any q_h in the global function space

$$V_h = \bigoplus_{k=1}^{K} V_h^k. \tag{4.5}$$

The discrete weak form then is

$$\int_\Omega \left(\frac{\partial q_h}{\partial t} + \frac{\partial f(q_h)}{\partial x} \right) v \, dx = 0, \tag{4.6}$$

with $q_h \in V_h$. We now choose the basis functions $\{\phi_i^k\}_{i=1,\dots,N_{bf}}$ on T_k as test functions v local to the element T_k. Since the basis functions vanish outside of T_k, this reduces the integration domain.

$$\int_{T_k} \left(\frac{\partial q_h}{\partial t} + \frac{\partial f(q_h)}{\partial x} \right) \phi_i^k \, dx = 0 \qquad i = 1, \dots, N_{bf} \tag{4.7}$$

So far, these integrals are fully element intern. We couple them to other elements, using integration by parts and replacing the flux in the boundary term with a numerical flux f^* similar to the one in finite volume methods.

$$\boxed{\int_{T_k} \frac{\partial q_h}{\partial t} \phi_i^k \, dx = \int_{T_k} f(q_h) \frac{d\phi_i^k}{dx} \, dx - \left[f^* \phi_i^k \right]_{x_l^k}^{x_r^k} \qquad i = 1, \dots, N_{bf}} \tag{4.8}$$

Now, we have two element intern integrals and a boundary term evaluated at the left and right element boundaries (x_l^k and x_r^k). This is the basic formulation for scalar discontinuous Galerkin methods in 1D (Knezevic, 2012).

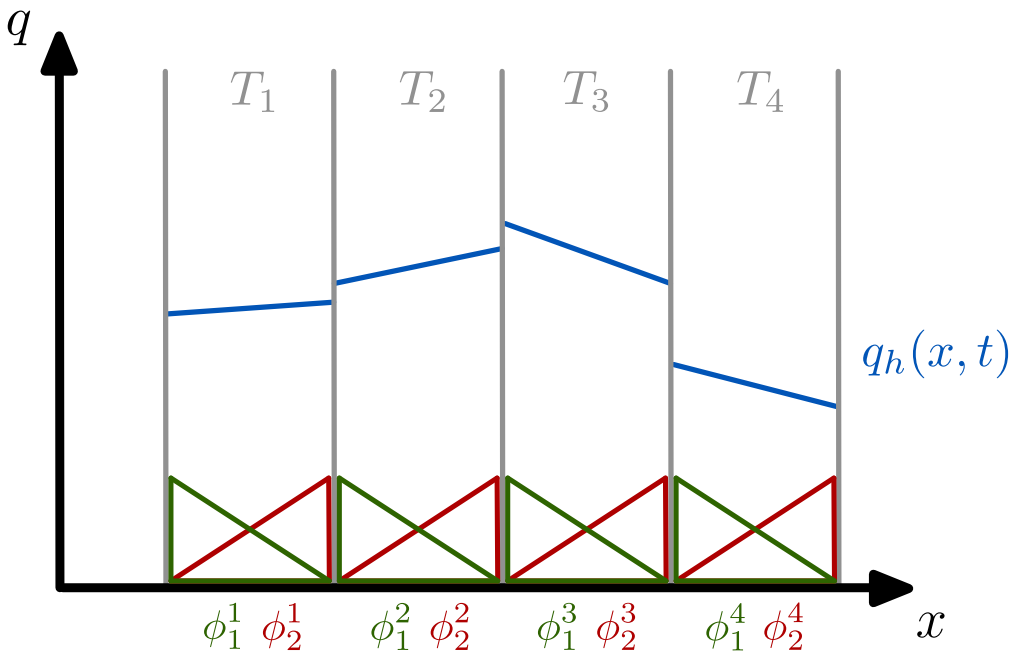

Figure 62: Function representation in DG. One can see two example basis functions in green and red, while the function q_h is in blue. As basis functions, we could, in principle, choose anything.

4.1.2 Relation to finite volume scheme

If we choose $N_p = 0$ (i.e. constant basis functions), this formulation reduces to

$$\int_{T_k} \frac{\partial q_h}{\partial t} \, dx = -\left[f^* \right]_{x_l^k}^{x_r^k} \qquad i = 1, \dots, N_{bf}, \tag{4.9}$$

which, after rearranging, becomes

$$\frac{d}{dt}\int_{T_k} q^h(x,t) = -(f^*(x_r^k) - f^*(x_l^k)). \tag{4.10}$$

Interestingly, this is equivalent to the finite volume scheme (see Eq. (3.7)). Whenever we will look at DG with $N_p = 0$ later, this will be equivalent to first-order FV (Knezevic, 2012).

4.1.3 Matrix assembly

To find an update formula for the solution coefficients, we substitute the basis expansion of q_h Eq. (4.3) into our DG formulation Eq. (4.8).

$$\sum_{j=1}^{N_{bf}} \frac{dQ_j^k(t)}{dt} \underbrace{\int_{T_k} \phi_j^k \phi_i^k \, dx}_{\mathbf{M}_{ij}^k} = \underbrace{\int_{T_k} f(q_h)\frac{d\phi_i^k}{dx}\, dx - \left[f^*\phi_i^k\right]_{x_l^k}^{x_r^k}}_{RHS_i^k} \quad i = 1, \ldots, N_{bf} \tag{4.11}$$

This can be written in matrix notation introducing the mass matrix $\mathbf{M}_{ij}^k$ and the right-hand side vector RHS_i^k (Knezevic, 2012)

$$\mathbf{M}_{ij}^k \frac{dQ_j^k(t)}{dt} = RHS^k(Q(t))_i, \tag{4.12}$$

where we use the Einstein summation convention. Assuming that the mass matrix is invertible, obtaining an ODE for the evolution of the solution coefficients is now straightforward.

$$\frac{dQ_j^k(t)}{dt} = (\mathbf{M}_{ij}^k)^{-1} RHS^k(Q(t))_i \tag{4.13}$$

For the temporal discretization, we will use a simple forward Euler scheme again (Knezevic, 2012).

$$\boxed{Q_j^{k,n+1} = Q_j^{k,n} + \Delta t(\mathbf{M}_{ij}^k)^{-1} RHS^k(Q^n, t^n)_i} \tag{4.14}$$

Note that this will limit the temporal convergence of our code to first-order, even if the spatial order is higher. When evaluating the L1 error convergence, later on, we have to look at the temporal and spatial order separately.

To use this scheme, we still need to calculate $\mathbf{M}_{ij}^k$ and RHS_i^k. To simplify this process, we map the element integrals from T_k to a reference element $\hat{T}$ (Knezevic, 2012). This is possible, since we will use the same basis functions on all elements. For our 1D case we can define an affine map between $x \in T_k$ and $\hat{x} \in \hat{T} \equiv [-1, 1]$ (see Fig. 63).

$$x = A_k \hat{x} + B_k \tag{4.15}$$

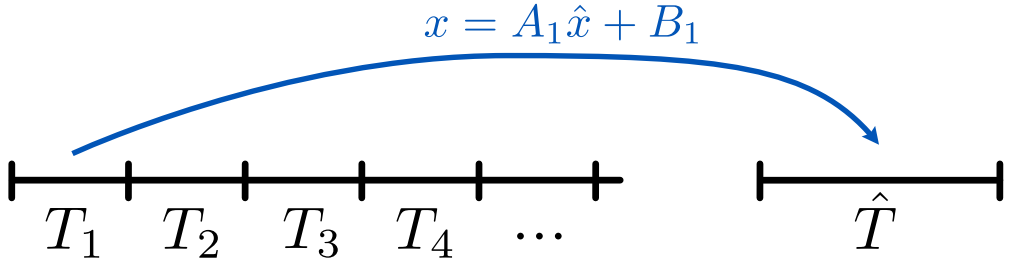

Figure 63: Relation between an element and the reference element.

We then define the basis functions on T_k by mapping the basis functions of $\hat{T}$.

$$\phi_i^k(x) \equiv \hat{\phi}_i(\hat{x}) \tag{4.16}$$

$$(\phi_i^k)'(x) = \frac{1}{A_k}(\hat{\phi}_i)'(\hat{x}) \tag{4.17}$$

The Mass matrix can then be rewritten to

$$\mathbf{M}_{ij}^k = \int_{T_k} \phi_i(x)\phi_j(x)\, dx = \int_{\hat{T}} \hat{\phi}_i(\hat{x})\hat{\phi}_j(\hat{x})\frac{dx}{d\hat{x}}\, d\hat{x}, \tag{4.18}$$

where for the affine map we have $\frac{dx}{d\hat{x}} = A_k$. One can now integrate to obtain the mass matrix. For simple cases like monomial basis functions $\hat{\phi}_i(\hat{x}) = \hat{x}^{i-1}$, this can be done directly. In contrast, for more complex basis functions, one can use integral approximations like Gauss quadrature (see section 4.3.2). For 1D, however, we will just use simple monomial basis functions. (Knezevic, 2012)

To calculate the right hand side vector (RHS), we first split it into RHS_{I} and RHS_{II} with

$$RHS_{\mathrm{I}}{}_i^k = \int_{T_k} f(q_h)\frac{d\phi_i^k}{dx}\, dx \tag{4.19}$$

$$RHS_{\mathrm{II}}{}_i^k = -\left[f^*\phi_i^k\right]_{x_l^k}^{x_r^k}. \tag{4.20}$$

For advection with velocity u, we can use $f(q_h) = uq_h$ to rewrite the RHS_{I} as

$$RHS_{\mathrm{I}}{}_i^k = \int_{T_k} f(q_h)\frac{d\phi_i^k}{dx}\, dx = \int_{T_k} uq_h\frac{d\phi_i^k}{dx}\, dx \tag{4.21}$$

$$= u\sum_{j=1}^{N_{bf}} Q_j^k \int_{T_k} \phi_j^k\frac{d\phi_i^k}{dx}\, dx = u\mathbf{S}_{ij}^k Q_j^k, \tag{4.22}$$

where we use the summation convention again and introduce the so-called advection matrix $\mathbf{S}_{ij}^k$ (Knezevic, 2012). We can write the advection matrix in terms of the reference element.

$$\mathbf{S}_{ij}^k = \int_{T_k} \phi_j^k\frac{d\phi_i^k}{dx}\, dx = \int_{\hat{T}} \hat{\phi}_j^k(\hat{x})\frac{d\hat{\phi}_i^k}{d\hat{x}}d\hat{x} \tag{4.23}$$

For a given set of basis functions, one can now directly calculate this.

Finally, we need to calculate RHS_{II}. Since it is a boundary term, we can evaluate the values directly.

$$RHS_{\mathrm{II}_i}^k = -\left[f^* \phi_i^k\right]_{x_l^k}^{x_r^k} = -f^*(x_r^k)\phi_i^k(x_r^k) + f^*(x_l^k)\phi_i^k(x_l^k) \tag{4.24}$$

For f^*, one has to use the numerical flux between the two states. For complex equations, one could employ Riemann solvers in that step, but for advection, we will use a simple upwind scheme. Using an advective flux $f(q_h) = uq_h$ to the right (i.e. $u > 0$) and taking the upwind side, leads to

$$RHS_{\mathrm{II}_i}^k = -uq_h(x_r^k)\phi_i^k(x_r^k) + uq_h(x_r^{k-1})\phi_i^k(x_l^k). \tag{4.25}$$

One would have to adapt this formula for advection in the other direction. After plugging the basis expansion of q_h (Eq. (4.3)) in, we get

$$RHS_{\mathrm{II}_i}^k = -u \sum_{j=1}^{N_{bf}} Q_j^k \underbrace{\phi_j^k(x_r^k)\phi_i^k(x_r^k)}_{(\mathbf{F_A})_{ij}^k} + u \sum_{j=1}^{N_{bf}} Q_j^{k-1} \underbrace{\phi_j^{k-1}(x_r^{k-1})\phi_i^k(x_l^k)}_{(\mathbf{F_B})_{ij}^k}. \tag{4.26}$$

One can rewrite this in matrix notation again

$$RHS_{\mathrm{II}_i}^k = -u(\mathbf{F_A})_{ij}^k Q_j^k + u(\mathbf{F_B})_{ij}^k Q_j^{k-1}, \tag{4.27}$$

where the matricies $\mathbf{F_A}$ and $\mathbf{F_B}$ can be calculated directly by plugging in values. (Knezevic, 2012)

Since we now have RHS_{I} and RHS_{II}, we can assemble the total right hand side vector RHS by addition

$$RHS_i^k = RHS_{\mathrm{I}_i}^k + RHS_{\mathrm{II}_i}^k = u\mathbf{S}_{ij}^k Q_j^k - u(\mathbf{F_A})_{ij}^k Q_j^k + u(\mathbf{F_B})_{ij}^k Q_j^{k-1} \tag{4.28}$$

and use it together with the mass matrix in our update scheme Eq. (4.14). Note that we have only assumed the mesh to be one dimensional and not required that the mesh is Cartesian. Regardless, we will use this formulation only for a 1D Cartesian mesh.

4.1.4 Boundary conditions

Boundary conditions work similar to the ones in a finite volume scheme. We implement them by simply setting the neighbors of the boundary cells to the correct values (Knezevic, 2012). In 1D, we implement an inflow boundary condition for the leftmost cell by setting its left neighbor value to q_{in}.

$$f^*(x_l^1) = f(q_{in}) \tag{4.29}$$

We will generally use $q_{in} = 0$. On the right side, we use an outflow boundary condition, which is already implemented naturally in the upwind scheme.

$$f^*(x_r^K) = f(q(x_r^K)) \tag{4.30}$$

4.2 Implementation of 1D advection

We will now implement the upwind advection scheme with monomial basis functions (see Fig. 64)

$$\hat{\phi}_i(\hat{x}) = \hat{x}^{i-1} \qquad \hat{x} \in [-1, 1] \tag{4.31}$$

on a 1D Cartesian grid, such that $N_p = N_{bf} - 1$ and $A_k = A$ for all elements.

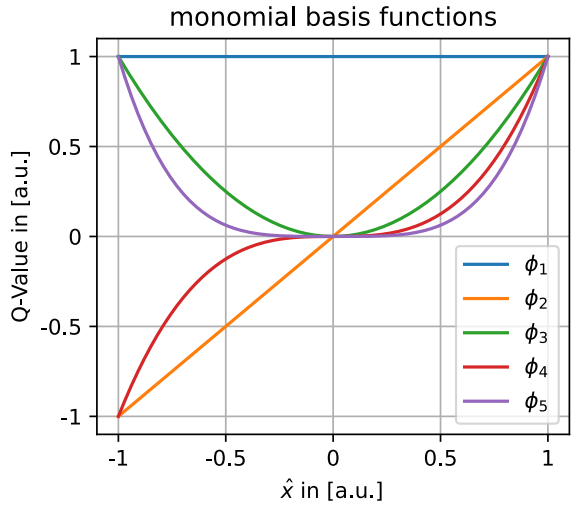

Figure 64: Monomial basis functions for $i = 1, 2, 3, 4, 5$.

The mass matrix can then be calculated manually and is

$$\mathbf{M}^k_{ij} = \int_{\hat{T}} \hat{\phi}_i \hat{\phi}_j \frac{dx}{d\hat{x}}\, d\hat{x} = \int_{[-1,1]} \hat{x}^{i-1}\hat{x}^{j-1} A\, d\hat{x} = \begin{cases} \frac{2A}{i+j-1} & \text{for } i+j \text{ even} \\ 0 & \text{for } i+j \text{ odd} \end{cases}. \tag{4.32}$$

In addition to that, we have $A_k = 1/N$. Similarly, we can directly calculate the advection matrix

$$\mathbf{S}^k_{ij} = \int_{\hat{T}} \hat{\phi}_j \frac{d\hat{\phi}_i}{d\hat{x}} d\hat{x} = \int_{[-1,1]} \hat{x}^{j-1} \cdot (i-1) \cdot \hat{x}^{i-2} d\hat{x} = \begin{cases} \frac{2(i-1)}{i+j-2} & \text{for } i+j \text{ odd} \\ 0 & \text{for } i+j \text{ even} \end{cases}. \tag{4.33}$$

For the matricies $(\mathbf{F_A})^k_{ij}$ and $(\mathbf{F_B})^k_{ij}$ we get

$$(\mathbf{F_A})^k_{ij} = \phi^k_j(x^k_r)\phi^k_i(x^k_r) = 1 \cdot 1 = 1 \tag{4.34}$$

$$(\mathbf{F_B})^k_{ij} = \phi^{k-1}_j(x^{k-1}_r)\phi^k_i(x^k_l) = 1 \cdot (-1)^i = (-1)^i. \tag{4.35}$$

With that, we have explicit versions of all the matrices and can use our formula for the right-hand side vector Eq. (4.28) in combination with the update scheme Eq. (4.14) for a fully specified scheme.

For the first implementation results, we look at an advected step function. We will call the value of q_h Q-Value. In Fig. 65, the higher order polynomials keep the structure of the initial condition better. This is especially true for smooth initial conditions.

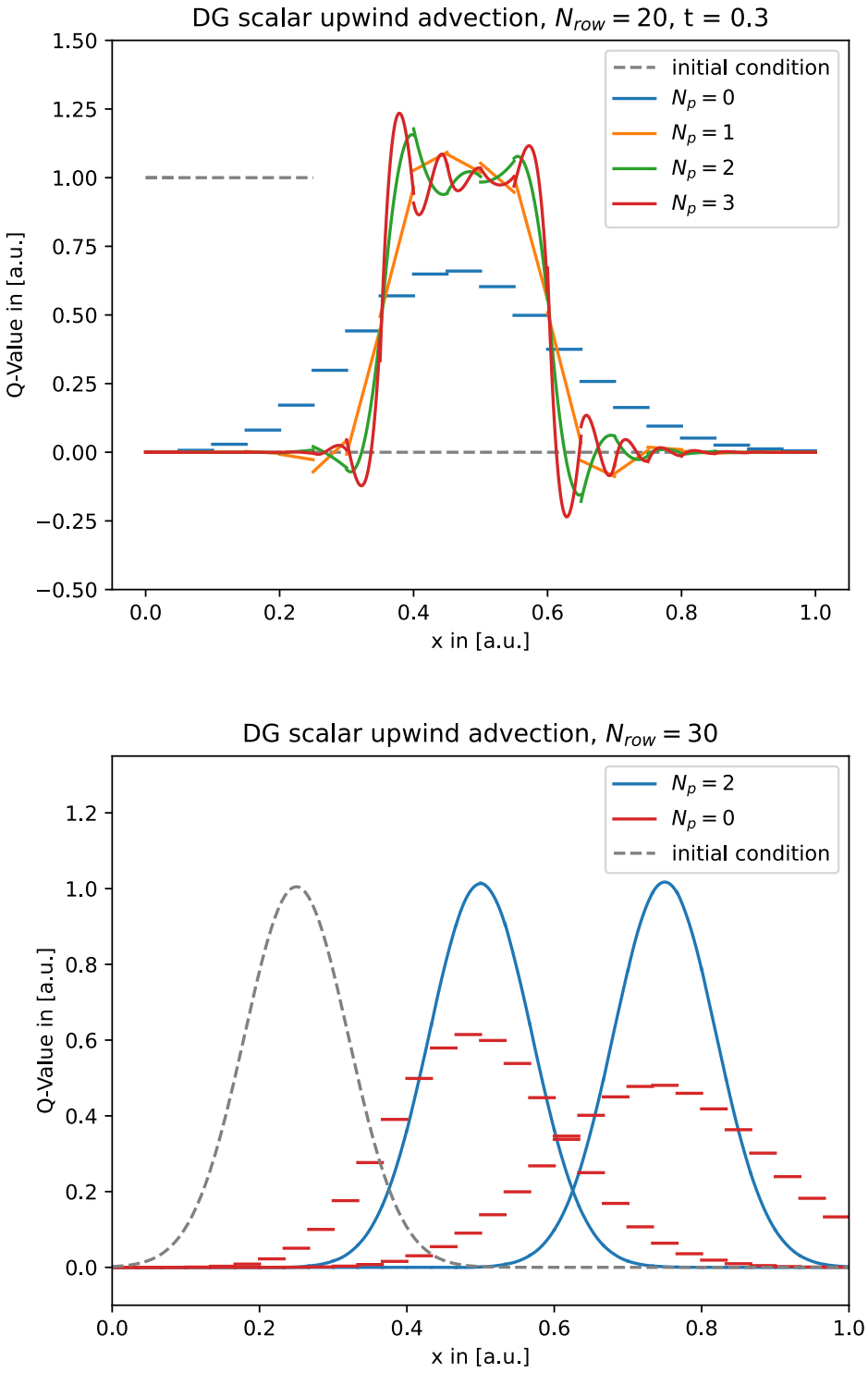

Figure 65: Top: Advected step function for $N = 20$ and different N_p. Bottom: Advected Gaussian for $N = 30$ and $N_p = 0, 2$. (Video)

However, for the discontinuous step function, the functions show oscillatory behavior near the discontinuities, which indicates that we should use slope limiting similar to higher-order FV methods.

4.2.1 Slope limiting

For slope limiting we consider linear approximations on each element. We compare the averaged slope on T_k $\langle \partial_x q_h^k \rangle$ with the averaged slope between $T_{k-1} \leftrightarrow T_k$ and the averaged slope between $T_k \leftrightarrow T_{k+1}$.

$$\tilde{q}_-^k = \frac{\langle q_h^k \rangle - \langle q_h^{k-1} \rangle}{\Delta x} \tag{4.36}$$

$$\tilde{q}_+^k = \frac{\langle q_h^{k+1} \rangle - \langle q_h^k \rangle}{\Delta x} \tag{4.37}$$

Here, Δx is the distance between the midpoint of two elements. We then do the following limiting procedure (Knezevic, 2012),(Schaal, 2016):

- If the three slopes have different signs, we set the slope on T_k to zero.
- If the three slopes have the same sign, we set the slope on T_k to the smallest in absolute value.

Mathematically, we realize this by

$$\langle \partial_x q_h^k \rangle_{SL} = \text{minmod}\Big(\langle \partial_x q_h^k \rangle, \theta \tilde{q}_+^k, \theta \tilde{q}_-^k \Big), \tag{4.38}$$

where θ is a free parameter to regulate how aggressive the slope limiting is. For $\theta = 1$, this slope limiter is TVD. The minmod function is given as

$$\text{minmod}(a,b,c) = \begin{cases} s \min(a,b,c) & \text{if } |s| = 1 \\ 0 & \text{else} \end{cases} \tag{4.39}$$

$$s = \frac{1}{3}\left(\frac{a}{|a|} + \frac{b}{|b|} + \frac{c}{|c|} \right). \tag{4.40}$$

We then compare the slope limited slope $\langle \partial_x q_h^k \rangle_{SL}$ with the original slope $\langle \partial_x q_h^k \rangle$. If both are equal, we change nothing on the element T_k and keep the higher-order (i.e. no limiting). If the slope limited version is different from the original, we set the function on the element T_k to

$$q_h^k = \langle q_h^k \rangle + (x - x_k) \cdot \langle \partial_x q_h^k \rangle, \tag{4.41}$$

where x_k is the midpoint of the element. By replacing the original function with a linear one, we throw away any higher-order information. This limits the solution so that no oscillations occur (Knezevic, 2012). However, there is large optimization potential here if one could keep the higher-order information somehow.
At the top of Fig. 66, we can see that the slope limiting effectively reduces any oscillations for the step function. Interestingly, though, the accuracy is quite dependent on θ. For the Gaussian on the bottom, we see, however, that the slope limiting reduces the accuracy, especially near the maximum. Therefore, future work should focus on improving oscillation detection to ensure that no slope limiter is active around a smooth peak.

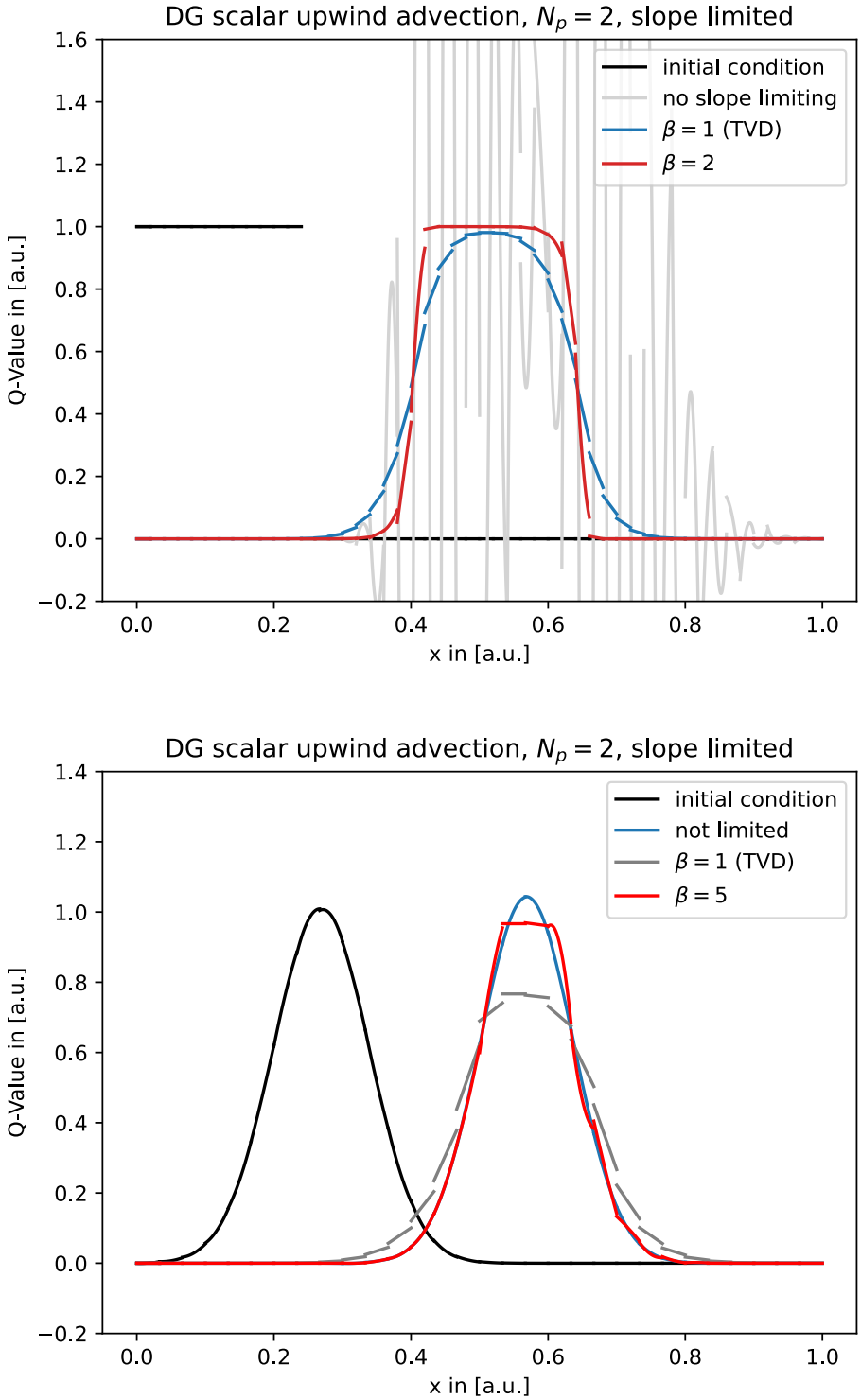

Figure 66: Top: Advected step function with $N_p = 2$ for different slope limiting and $N = 50$ (Video). Bottom: Advected Gaussian with $N_p = 2$ for different slope limiting and $N = 30$.

4.3 Generalization to two dimensions

Going into the second dimension, the formulation of DG generally stays the same as for 1D (Knezevic, 2012). Eq. (4.8) then becomes:

$$\boxed{\int_{T_k} \frac{\partial q_h}{\partial t} \phi_i^k \, d\vec{x} = \int_{T_k} \vec{f}(q_h) \cdot \vec{\nabla} \phi_i^k \, d\vec{x} - \int_{\partial T_k} \phi_i^k \vec{f}^* \cdot \vec{n} \, ds \quad i = 1, \ldots, N_{bf}} \tag{4.42}$$

The only things that change are the use of 2D element integrals instead of 1D, the replacement of the partial derivative with a gradient, and the boundary term becoming a line integral (Knezevic, 2012). Using the basis representation of q_h Eq. (4.3) and the Einstein summation convention, we can rewrite the integral for advection (i.e. $\vec{f}(q_h) = \vec{u} q_h$) to

$$\frac{\partial Q_j^k}{\partial t} \underbrace{\int_{T_k} \phi_j^k \phi_i^k \, d\vec{x}}_{\mathbf{M}_{ij}^k} = Q_j^k \vec{u} \cdot \underbrace{\int_{T_k} \phi_j^k \vec{\nabla} \phi_i^k \, d\vec{x}}_{\vec{\mathbf{S}}_{ij}^k = ((S_{ij}^k)_x, (S_{ij}^k)_y)} - \int_{\partial T_k} \phi_i^k \vec{f}^* \cdot \vec{n} \, ds \quad i = 1, \ldots, N_{bf}. \tag{4.43}$$

The main difference to the 1D version is that the advection matrix $\vec{\mathbf{S}}_{ij}^k$ is now a vector consisting of two matrices because of the gradient.

To calculate the integrals more efficiently, we will map them on a reference element again (Knezevic, 2012). We will use a Cartesian mesh with $N_{row} \times N_{row}$ cells on $\Omega \equiv [0, 1]^2$ and therefore use $\hat{T} \equiv [-1, 1]^2$ as a reference element. The affine map between T_k and $\hat{T}$ is given as

$$\vec{x} = \underbrace{\frac{1}{2N_{row}} \mathbb{1}}_{\mathbf{A_k}} \hat{\vec{x}} + \vec{s}_k, \tag{4.44}$$

where $\mathbb{1}$ is the unit matrix and $\vec{s}_k$ the seed on element T_k (see Fig. 67). The scaling factor A_k from 1D is now a matrix $\mathbf{A_k}$ (Schaal, 2016). For other meshes, like a triangular mesh, one could define a different map and reference element in this step and would get a 2D formulation for another mesh type (Knezevic, 2012). Also, this is the exact part where it gets interesting for a Voronoi mesh. Since a Voronoi mesh has cells with different numbers of edges, it is impossible to define only one reference element and an affine map towards it. Different approaches to this will be discussed in the outlook (see Sect. 5).

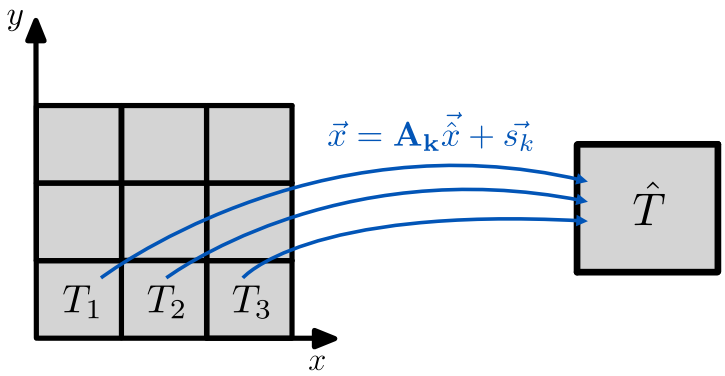

Figure 67: Affine maps from different T_k to the reference element $\hat{T}$.

Returning to the 2D Cartesian mesh, we also define our basis functions on T_k using the reference basis functions.

$$\phi_i^k(x) = \hat{\phi}_i(\hat{x}) \tag{4.45}$$

$$\vec{\nabla}\phi_i^k(x) = (\mathbf{A_k}^T)^{-1}\hat{\phi}(\hat{x}) = 2N_{row}\vec{\nabla}\hat{\phi}(\hat{x}) \tag{4.46}$$

We can now use the reference basis functions to calculate $\mathbf{M}_{ij}^k$ and $\vec{\mathbf{S}}_{ij}^k$ (Knezevic, 2012).

$$\mathbf{M}_{ij}^k = \int_{T_k} \phi_i^k \phi_j^k \, d\vec{x} = \det(\mathbf{A_k}) \int_{\hat{T}} \hat{\phi}_i \hat{\phi}_j \, d\vec{\hat{x}} = \frac{1}{4N_{row}} \int_{[-1,1]^2} \hat{\phi}_i \hat{\phi}_j \, d\hat{x} d\hat{y} \tag{4.47}$$

$$\vec{\mathbf{S}}_{ij}^k = \int_{T_k} \phi_j^k \vec{\nabla} \phi_i^k \, d\vec{x} = \frac{1}{2N_{row}} \int_{[-1,1]^2} \hat{\phi}_j \vec{\nabla} \hat{\phi}_i \, d\hat{x} d\hat{y} \tag{4.48}$$

Next, we calculate the third part of Eq. (4.43), using the reference basis functions. We use, that the line integral can be split into four straight lines on a Cartesian mesh.

$$\int_{\partial T_k} \phi_i^k \vec{f}^* \cdot \vec{n} \, ds = \frac{1}{2N_{row}} \Bigg(- \int_1 \hat{\phi}_i(-1,y) f_x^*(-1,y) \, dy + \int_2 \hat{\phi}_i(x,1) f_y^*(x,1) \, dx \tag{4.49}$$

$$+ \int_3 \hat{\phi}_i(1,y) f_x^*(1,y) \, dy - \int_4 \hat{\phi}_i(x,-1) f_x^*(x,-1) \, dx \Bigg) \tag{4.50}$$

Replacing the flux with $\vec{f}^* = \vec{u} q_h$ and using the basis representation of q_h on the upwind side plus summation convention leads to:

$$\int_{\partial T_k} \phi_i^k \vec{f}^* \cdot \vec{n} \, ds = \frac{1}{2N_{row}} \Bigg(-u_x Q_j^{k-1} \underbrace{\int_{-1}^{1} \hat{\phi}_i(-1,y) \hat{\phi}_j(-1,y) \, dy}_{(\mathbf{F_A})_{ij}^k} + u_y Q_j^k \underbrace{\int_{-1}^{1} \hat{\phi}_i(x,1) \hat{\phi}_j(x,1) \, dx}_{(\mathbf{F_B})_{ij}^k} \tag{4.51}$$

$$+ u_x Q_j^k \underbrace{\int_{-1}^{1} \hat{\phi}_i(1,y) \hat{\phi}_j(1,y) \, dy}_{(\mathbf{F_C})_{ij}^k} - u_y Q_j^{k-N_{row}} \underbrace{\int_{-1}^{1} \hat{\phi}_i(x,-1) \hat{\phi}_j(x,-1) \, dx}_{(\mathbf{F_D})_{ij}^k} \Bigg) \tag{4.52}$$

Here, we have implicitly assumed that for the velocity $\vec{u}$, we have $u_x > 0$ and $u_y > 0$. These integrals can be calculated directly for simple basis functions or using approximation methods for more complex cases. Substituting this into Eq. (4.43), gives us an update scheme for 2D Cartesian upwind advection from timestep n to timestep $n+1$.

$$\boxed{Q_j^{k,n+1} = Q_j^{k,n} + \Delta t (\mathbf{M})^{-1} \Big(\vec{u} \cdot \vec{\mathbf{S}} Q_j^k + \frac{1}{2N_{row}} \Big[u_x \mathbf{F_A} Q_j^{k-1} - u_y \mathbf{F_B} Q_j^k - u_x \mathbf{F_C} Q_j^k + u_y \mathbf{F_D} Q_j^{k-N_{row}} \Big] \Big)}$$

One can now update the value Q_j^k based on itself and its direct neighbors on the upwind side. The calculation of many integrals in the derivation suggests that the algorithm is computationally intensive. However, these integrals only have to be calculated once for an entire simulation run because they do not change over time and are the same for every element on a Cartesian mesh. The updating scheme then consists of a few matrix multiplications, which is quite efficient (Knezevic, 2012).

4.3.1 Legendre polynomials

For the basis functions, we follow (Schaal, 2016) and use products of Legendre polynomials. They are the solution to Legendre's differential equation.

$$\frac{d}{d\xi}\left[(1-\xi^2)\frac{d}{d\xi}P_n(\xi)\right] + n(n+1)P_n(\xi) = 0 \qquad n \in \mathbb{N}_0 \tag{4.53}$$

Given the first two polynomials $P_0(\xi) = 1$ and $P_1(\xi) = \xi$, higher order polynomials can be computed recursively.

$$P_n(\xi) = \frac{1}{n}\Big[(2n-1)\xi P_{n-1} - (n-1)P_{n-2}(\xi)\Big] \tag{4.54}$$

The polynomials are pairwise orthogonal on $[-1, 1]$.

$$\int_{-1}^{1} P_i(\xi)P_j(\xi)\, d\xi = \begin{cases} 0 & \text{if } i \neq j \\ \frac{2}{2i+1} & \text{if } i = j \end{cases} \tag{4.55}$$

This is interesting for us since the mass matrix will be diagonal, which makes inverting it straightforward. As our basis functions, we will use products of the first three Legendre polynomials.

$$P_0(\xi) = 1,\ P_1(\xi) = \xi,\ P_2(\xi) = \frac{1}{2}(3\xi^2 - 1) \tag{4.56}$$

A simple $N_p = 0$ basis then is $\{P_0\}$, while for an $N_p = 1$ basis we have $\{P_0, P_1(x), P_1(y)\}$. The $N_p = 2$ basis finally is $\{P_0, P_1(x), P_1(y), P_1(x)P_1(y), P_2(x), P_2(y)\}$ (see Fig. 68).

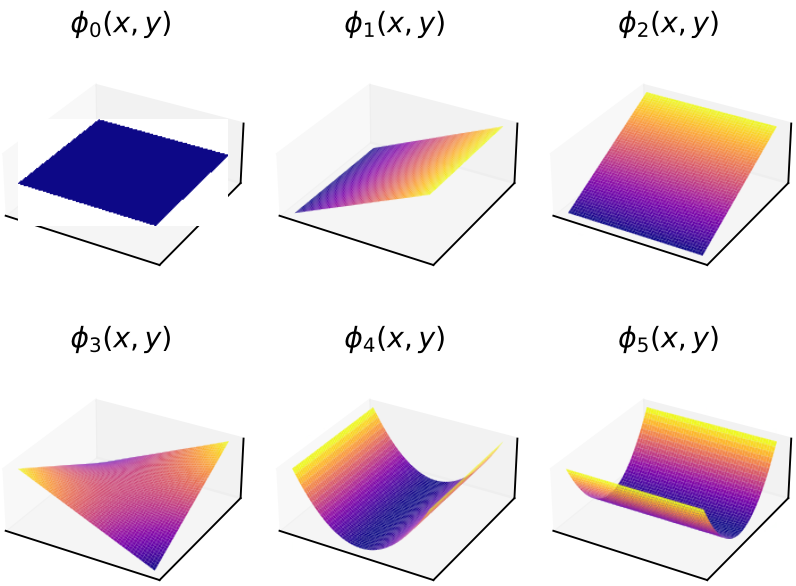

Figure 68: The basis functions of the $N_p = 2$ basis.

4.3.2 Gauss quadrature

There are many integrals that we need to calculate in the DG scheme. So far, we have done this analytically. Another way to calculate the integrals is the Gauss quadrature. Generally, a quadrature rule approximates an integral by a sum of the function value at quadrature points $\vec{\xi}_i^{2D}$, weighted with ω_i^{2D} (Schaal, 2016).

$$\int f(x) \approx \sum_i \omega_i^{2D} f(\vec{\xi}_i^{2D}) \tag{4.57}$$

There exist many different quadrature rules. A 1D quadrature rule with n points is exact for polynomials up to order $2n-1$ (Knezevic, 2012). We will use a 3×3 point quadrature rule on the $[-1,1]^2$ square in 2D for the next section (Schaal, 2016). The weights and quadrature points are given in Fig. 69.

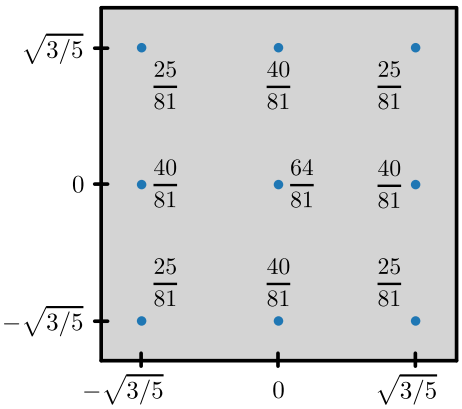

Figure 69: Gauss quadrature points and weights for 3×3 rule on $[-1,1]^2$.

4.3.3 Proper initial conditions

For the 1D case, we manually set the initial conditions by setting the coefficients. This works to confirm that the scheme roughly works. However, since we want to look at the L1 error for the 2D scheme later on, we need to implement proper initial conditions. By that, we mean initial conditions

$$q_h(\vec{x},0)\Big|_{T_k} = \sum_{j=1}^{N_{bf}} Q_j^k(0)\phi_j^k(\vec{x}), \tag{4.58}$$

that minimize the L^2 error relative to a function $q(x,t)$ (Knezevic, 2012). The necessary condition for that is

$$\begin{aligned} 0 &= \frac{\partial}{\partial Q_j^k(0)} \int_{T_k} \Big(q_h(\vec{x},0) - q(\vec{x},0)\Big)^2 \, d\vec{x} \\ &= 2 \int_{T_k} \left(\sum_l Q_l^k \phi_l^k - q(\vec{x},0)\right) \phi_j^k \, d\vec{x} \\ &= 2 \cdot \left(Q_j^k \int_{T_k} \phi_j^k \phi_j^k \, d\vec{x} - \int_{T_k} q(\vec{x},0)\phi_j^k \, d\vec{x}\right), \end{aligned}$$

where we used in the second step, that the basis functions are orthogonal (Schaal, 2016). Next, we change into the reference element coordinates and identify an integral $\tilde{M}_{jj}$, proportional to the diagonal elements of the mass matrix.

$$Q_j^k \underbrace{\int_{\hat{T}} \hat{\phi}_j \hat{\phi}_j \, d\vec{\hat{x}}}_{\tilde{\mathbf{M}}_{ij}} = \int_{\hat{T}} q(\vec{x}(\vec{\hat{x}}), 0) \hat{\phi}_j(\vec{\hat{x}}) \, d\vec{\hat{x}}$$

We can rewrite this to obtain a formula for the initial conditions.

$$Q_j^k(t=0) = \frac{1}{\tilde{\mathbf{M}}_{jj}} \int_{\hat{T}} q(\vec{x}(\vec{\hat{x}}), 0) \hat{\phi}_j(\vec{\hat{x}}) \, d\vec{\hat{x}}$$

Since we want to keep this part flexible for arbitrary initial conditions $q(\vec{x}, 0)$, we use Gauss quadrature instead of manually calculating the integral (Schaal, 2016). This leads to

$$Q_j^k(t=0) = \frac{1}{\tilde{\mathbf{M}}_{jj}} \sum_{i=1}^{9} \omega_i^{\text{2D}} q(\vec{x}(\vec{\xi}_i^{\text{2D}}), 0) \hat{\phi}_j(\vec{\xi}_i^{\text{2D}}, 0).$$

We use the quadrature points and weights for the 2D square mentioned in the previous section. Only at this point of the derivation we assume a 2D Cartesian mesh. For a 2D Gaussian $q(\vec{x}, 0)$, one can see a 1D slice of the optimal initial conditions for different N_p in Fig. 70.

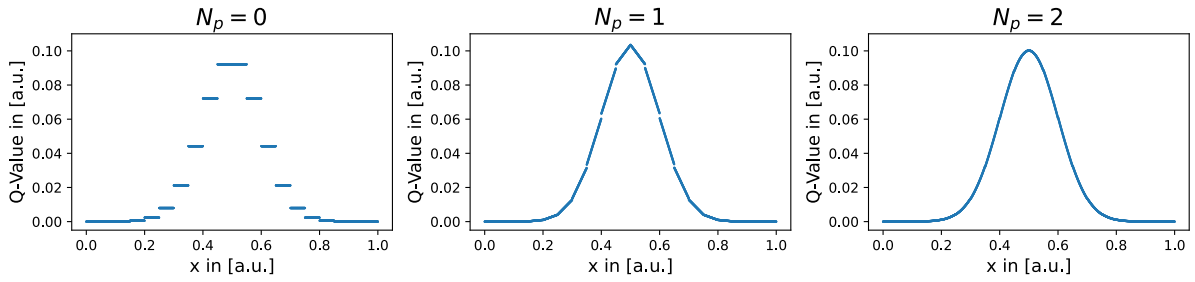

Figure 70: Proper initial conditions for a Gaussian minimizing the L^2 error. This image is a slice through a 20×20 Cartesian mesh.

4.4 Implementation of 2D advection

Plotting the function q_h in 2D, given the coefficients Q_j^k, is not as simple as in finite volume. Instead of a color that we can assign to the cell average, we now have to assign colors according to the functions inside each element. For that, we subsample every element with a further Cartesian grid to approximate the function. The size of the subsampling grid can then be changed as needed.
We will test the implementation with an initial square function and an initial Gaussian. The advection velocity is set to $\vec{u} = (0.5, 0.5)$ for both. In Fig. 71, an initial square function is advected over the mesh for different N_p without a slope limiter. As one can see, the higher-order schemes keep the shape of the square function better.

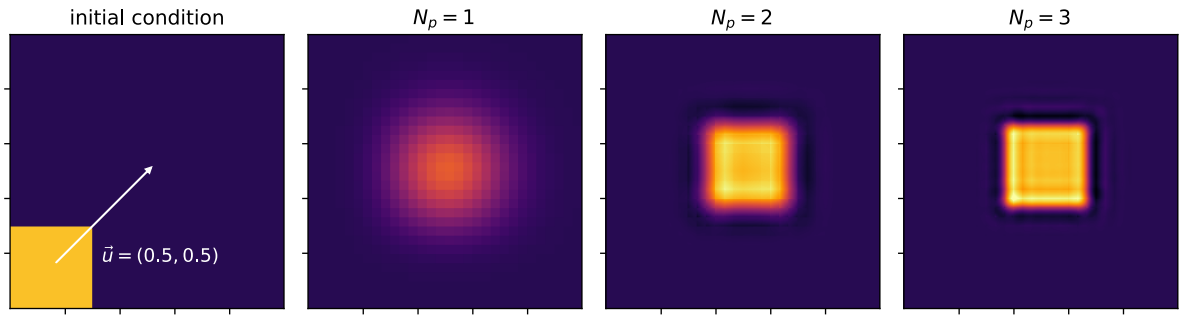

Figure 71: Advected square for different N_p without slope limiting on a 30×30 mesh.

However, as in 1D, we observe oscillations near the discontinouities. The slope limiting is conceptually the same as in 1D. We limit both dimensions separately. The same initial square advected over the mesh with slope limiting looks like Fig. 72.

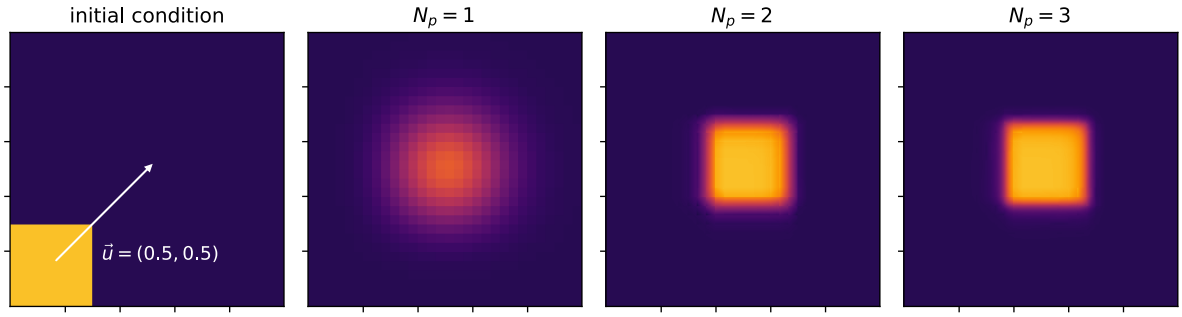

Figure 72: Advected square for different N_p with slope limiting on a 30×30 mesh. Using $\theta = 2$.

The oscillations are gone now, and the advected squares, for higher N_p, keep the initial shape better. However, there is no huge improvement between $N_p = 1$ and $N_p = 2$. We will return to this later in the L1 error analysis.

Next we look at a Gaussian peak without slope limiting.

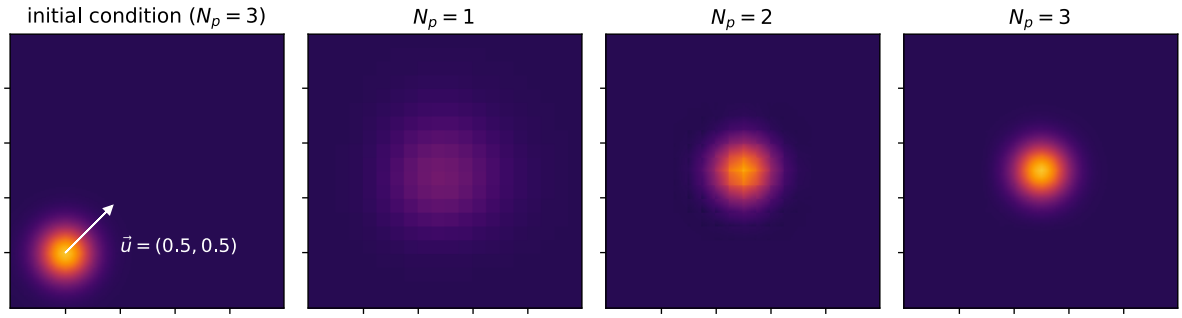

Figure 73: Advected Gaussian for different N_p without slope limiting on a 20×20 mesh. The initial condition is the one for $N_p = 3$.

In Fig. 73, the shape of the Gaussian is kept much better for higher-order schemes. Also, a clear improvement from $N_p = 2$ to $N_p = 3$ is visible. Therefore, it seems that just as in finite volume methods, the slope limiting reduces the convergence to first-order, while the methods on smooth solutions generally can be higher-order.

4.4.1 L1 error convergence

To check this, we look at the L1 error convergence. Before doing so, we need to redefine the L1 error for the elements

$$\mathrm{L1} = \frac{1}{N} \sum_{i=0}^{N} \frac{1}{A_i} \int_{T_i} |q(\vec{x}, t) - q_h(\vec{x}, t)| \, d\vec{x}, \tag{4.59}$$

where A_i is the area of element T_i. This definition returns the average error per element.

We start by looking at the L1 error over resolution for an initial square, as in Fig. 72 with slope limiting. As shown in Fig. 74, the higher-order methods have less than half the error of the first-order method. However, there is no huge improvement from second- to third-order and the scaling overall stays below first-order convergence. This is similar to what we have observed for the FV method scaling, where the slope limiting around discontinuities limited the convergence order to below one.

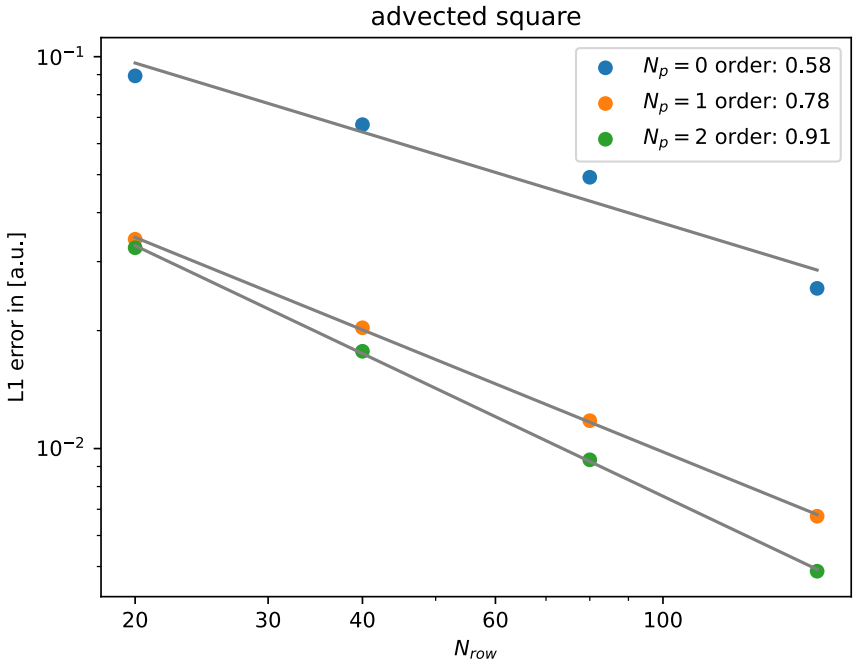

Figure 74: L1 error for advected square. Limiting reduces the convergence order to below one.

To check if we can achieve higher-order convergence without slope limiting on a smooth solution, we look at an initial Gaussian. To observe the higher-order convergence, we have to remember that we have implemented a scheme with a high-order polynomial reconstruction in space and a simple forward Euler time integration. Therefore, we have

to look at the spatial and time convergence separately[2]. The total L1 error we observe is made up of the error from correctly representing the function in space and from correctly solving the time integration. If now, the spatial L1 error converges with e.g. third-order and the Euler scheme with first-order only, we would observe a mix of those two. For spatial convergence, we look at the L1 error for different mesh resolutions, with extremely small time-stepping ($\Delta t \approx \Delta t_{CFL}/10^4$). The error of the time integration then is negligible and we observe only the spatial L1 error component.

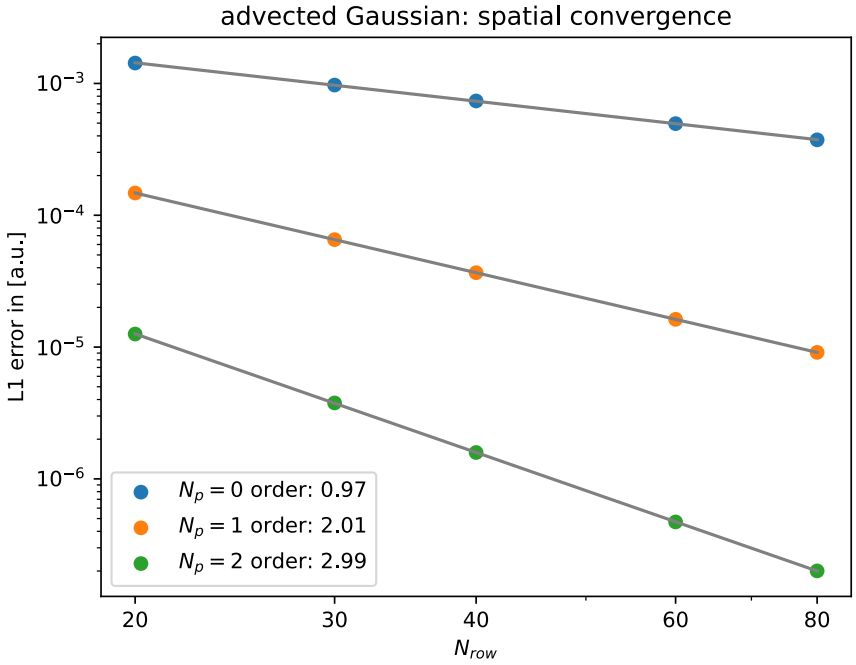

Figure 75: Spatial L1 error for advected Gaussian.

In Fig. 75, we observe higher-order spatial convergence just as expected for a first-, second- and third-order method. Also, in absolute value, there is a huge difference in accuracy. This plot, therefore, shows that the implemented DG scheme can indeed reach higher-order in a very systematic way.

For observing the convergence of the time integration error, we want to choose the resolution and timestep so that the spatial error is way smaller than the time integration error and thus negligible. If we then vary the time-stepping, we can observe the time integration error convergence.
For the second- and third-order methods, we can use that the spatial error converges faster than the time integration error. Increasing the resolution at constant CFL number therefore reduces the ratio between time integration and spatial error. For really high resolution the spatial error then is negligible compared to the time integration error and we can observe the time integration error convergence by varying the number of timesteps. This is not possible for the first-order method, since both the spatial and time integration errors converge with the same order. Increasing the resolution at constant CFL number,

[2]This was not necessary for the square since the limited DG and the forward Euler scheme are both first-order.

then does not change the ratio between the two errors. And since we can not choose arbitrary large CFL numbers (because the time integration then becomes unstable), there is no way to eliminate the spatial error without reducing the time error in the same way. Thus, the spatial error will always dominate the total error for the first-order scheme.

We, therefore, can only observe the time convergence for the higher-order schemes as shown in Fig. 76. As one can see, the time integration error for $N_p = 1$ and $N_p = 2$ is roughly similar and converges with first-order.

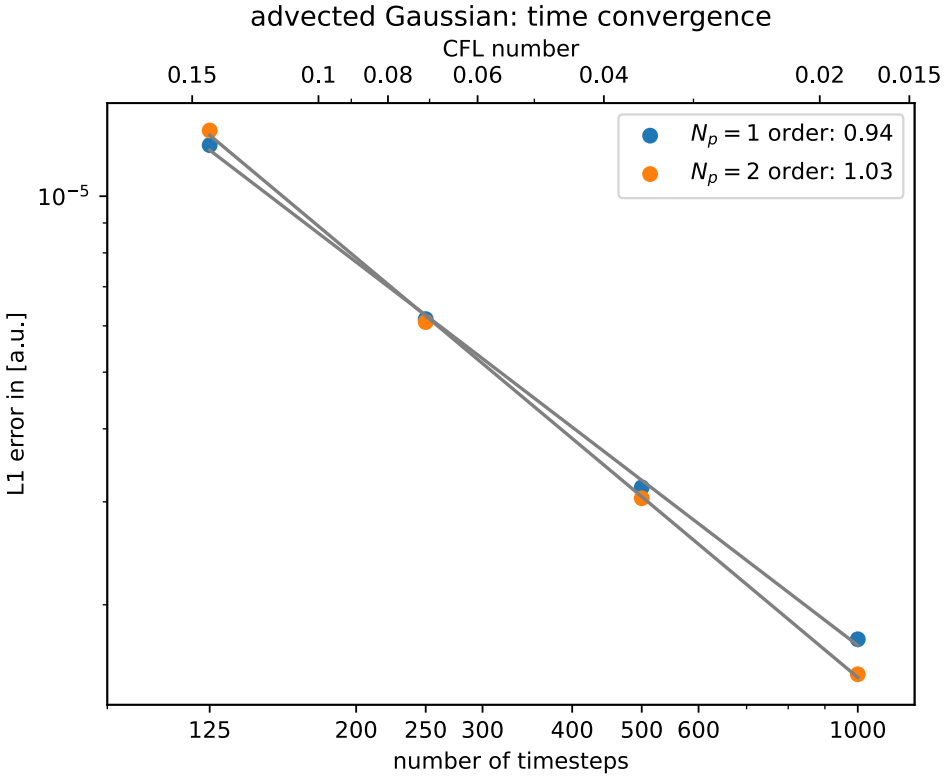

Figure 76: L1 error for different time-stepping of advected Gaussian on $N = 62500$ elements. Total simulation time $t = 0.1$.

This is expected since both methods use the same forward Euler time integration. Since the total L1 error will always include the time integration and spatial reconstruction components, the overall convergence will be determined by the lowest convergence (i.e., first-order from the forward Euler scheme). To actually benefit from the higher-order DG schemes, one would have to improve the time integration to something like a higher-order Runge-Kutta integrator (Springel et al., 2020), which should be addressed in future work.

Recap: Discontinous Galerkin Methods

This section introduced the general theory and implementation for scalar upwind advection on 1D and 2D Cartesian grids. In 1D, we demonstrated the need for slope limiting for discontinuous functions and showed that smooth solutions are represented very well by the scheme. We then demonstrated that slope limiting solves the problem of oscillatory behavior near discontinuities at the expense of significantly reducing accuracy for smooth solutions. Better oscillation detection, therefore, would be important in future work. In 2D, we quantitatively studied the convergence of the discontinuous square initial condition compared to the smooth Gaussian. Here, we demonstrated that similar to the finite volume scheme, slope limiting reduces the convergence to first-order, while higher-order is possible for smooth solutions without slope limiting. There, we also observed that the L1 error consists of a time integration and a spatial representation error. Even though we demonstrated systematic higher-order spatial convergence for smooth solutions, the time integration error overall dominates and reduces the convergence of the scheme for higher resolution to first-order. Therefore, higher-order time integration schemes should be included in the future. Although not used, most of these concepts directly transfer to solving other equations, like Euler or shallow water equations.

5 Summary and outlook

Summary

In this thesis, we have explored two major methods for numerically solving hydrodynamic equations: the finite volume method with a second-order MUSCL scheme and the discontinuous Galerkin method. First, we focussed on generating a mesh to simulate the equations. For that, we looked at Voronoi meshes, which are especially useful for astrophysics. There, we demonstrated that the presorted point insertion algorithm scales optimally with $\mathcal{O}(n \log n)$ and that the memory grows linearly with mesh size. We also implemented a way to relax the mesh and custom unstructured internal boundaries, like an airfoil or a circle.
We then implemented a Godunov-type finite volume scheme with an approximate Riemann solver. In the next step, we improved the algorithm to a second-order MUSCL scheme by using piece-wise linear approximations to solve the Riemann problems and employing slope limiting following (Springel, 2010), (Springel et al., 2020). For the implementation results, we looked at the advection equation with an upwind scheme and observed numerical diffusion scaling as predicted in theory (Dullemond and H.H.Wang, 2009). We also demonstrated that the scheme is conservative and that the Cartesian mesh shows an angular dependence in numerical diffusion. The shallow water equations were solved by a first- and second-order Godunov-type scheme, which we verified using the analytically known dam break problem (Delestre et al., 2013). In addition to that, we verified the solver for smooth solutions by comparing it to the simulation results of a different code (Cortild, 2023). Looking at the convergence, we observed that slope limiting around discontinuities decreases the convergence to first-order, even for higher-order methods. For the smooth solution, instead, we could show convergence better than first-order. We then used the solver to simulate shallow water waves around custom inner boundaries shaped, for example, like continents, to demonstrate potential use cases of the code. After verifying the solver for the Euler equations using the shock tube test, we demonstrated that various known hydrodynamic phenomena also occur in the code. We started by reproducing the Kelvin-Helmholtz instability and the Rayleigh-Taylor instability. In both, we observed that the size of the smallest eddies in ideal hydrodynamics is a phenomenon that strongly depends on mesh resolution. After reproducing a 2D Riemann problem from (Kurganov and Tadmor, 2002), we observed vortex shedding around a circle and the flow around an airfoil, generating lift.
For implementing a discontinuous Galerkin scalar upwind advection scheme, we derived the formulation on a Cartesian mesh in 1D and 2D following (Knezevic, 2012) and (Schaal, 2016). Looking at the implementation results, we again observed the need for slope limiting and implemented this as well. Afterwards, we looked at the convergence of the scheme and observed that, just as in FV, the convergence is limited to first-order when using a slope limiter near discontinuities. For a smooth solution, we split the L1 error into a spatial and time component and showed that the DG scheme shows promising higher-order spatial convergence. However, the time convergence is still first-order due to the forward Euler time integration. Overall, the scaling of the DG code clearly shows potential for future applications.

Outlook

There exist many different ways to improve the current version of the code, and many questions need to be studied further. For example, one could look at why the MUSCL scheme scales worse than second-order for smooth solutions. One possible idea here might be that although the solution is quite smooth, the slope limiting is still active in some areas, reducing overall accuracy. To figure this out, one could check whether the number of cells where the slope limiter was active during a step is significant. For the DG scheme, the first obvious improvement would be to change to a higher-order time integration. At the moment, the total L1 error is dominated by the time integration error, which reduces the benefit of the higher-order spatial reconstruction. In addition, one could implement higher-order slope limiting to prevent the error around discontinuities, scaling with first-order only. This could be relevant, especially when simulating astrophysical objects with many shocks.

In future work, one could generalize the DG scheme to also solve the Euler equations. This has been done by e.g. (Schaal, 2016). For real applications, the code should also be extended to handle three dimensions. This would require a different Voronoi mesh generation since we rely on going around the cell, which we can not do in 3D. Three-dimensional simulations also require way more computational power, which is why a possible future focus could be on parallelizing the code to run on multiple cores or nodes. Another interesting aspect could be to focus on how to make the mesh move with the flow. This has been done for finite volume, e.g. in (Springel, 2010). For DG, however, the ideal approach to a moving mesh code is still active research, and different approaches exist. A difficulty with a Voronoi mesh is that one can not define a reference element easily since Voronoi cells can have different numbers of edges. This becomes especially difficult if the mesh moves with the flow and a cell changes the number of edges between a timestep (Gaburro et al., 2020). Approaches to this include using centroidal basis functions (Mocz et al., 2013), extending the cells into the time dimension (Gaburro et al., 2020), and decomposing the Voronoi cell into tetrahedra (Walter Boscheri et al., 2022). In the long term, one could focus on implementing and maybe even improving one of the different approaches.

Acknowledgements

Working on this project has been really exciting and I want to thank Dylan, Chris, and the entire group for the awesome last few months. It has been a pleasure working with you all! It's not a given that you dedicated so many hours of your time to me and my project. I really appreciate that!

References

J. D. Anderson. Fundamentals of aerodynamics, 2007.

M. Bartelmann. *Theoretical Astrophysics: An Introduction*. Lecture Notes. Heidelberg University publishing, März 2021. doi: 10.17885/heiup.822. https://heiup.uni-heidelberg.de/catalog/book/822.

D. Cortild. Well-balanced schemes for shallow water equations. 11 2023. doi: 10.13140/RG.2.2.13232.94723. https://www.researchgate.net/publication/375552742_Well-Balanced_Schemes_for_Shallow_Water_Equations.

C. Dawson and C. M. Mirabito. The shallow water equations, 2008. https://users.oden.utexas.edu/~arbogast/cam397/dawson_v2.pdf.

O. Delestre, C. Lucas, P.-A. Ksinant, F. Darboux, C. Laguerre, T. N. T. Vo, F. James, and S. Cordier. SWASHES: a compilation of Shallow Water Analytic Solutions for Hydraulic and Environmental Studies. *International Journal for Numerical Methods in Fluids*, 72 (3):269–300, May 2013. doi: 10.1002/fld.3741. https://hal.science/hal-00628246.

P. C. Duffell and A. I. MacFadyen. Tess: A relativistic hydrodynamics code on a moving voronoi mesh. *The Astrophysical Journal Supplement Series*, 197(2):15, Nov. 2011. ISSN 1538-4365. doi: 10.1088/0067-0049/197/2/15. http://dx.doi.org/10.1088/0067-0049/197/2/15.

C. Dullemond and H.H.Wang. Lecture on numerical fluid dynamics, 2009. https://www.ita.uni-heidelberg.de/~dullemond/lectures/num_fluid_2009/index.shtml.

E. Gaburro, W. Boscheri, S. Chiocchetti, C. Klingenberg, V. Springel, and M. Dumbser. High order direct arbitrary-lagrangian-eulerian schemes on moving voronoi meshes with topology changes. *Journal of Computational Physics*, 407:109167, Apr. 2020. ISSN 0021-9991. doi: 10.1016/j.jcp.2019.109167. http://dx.doi.org/10.1016/j.jcp.2019.109167.

S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Mat. Sb., Nov. Ser.*, 47:271–306, 1959.

Hydrotec. Hydrodynamic-numerical software for 2d simulation in the areas of flood protection, waterway development, and heavy rain prevention. https://www.hydrotec.de/software/hydroas/.

G. D. Joshi, A. Pillepich, D. Nelson, F. Marinacci, V. Springel, V. Rodriguez-Gomez, M. Vogelsberger, and L. Hernquist. The fate of disc galaxies in illustristng clusters. *Monthly Notices of the Royal Astronomical Society*, 496(3):2673–2703, June 2020.

ISSN 1365-2966. doi: 10.1093/mnras/staa1668. http://dx.doi.org/10.1093/mnras/staa1668.

D. D. Knezevic. Lecture notes on computatuonal fluid dynamics, applied mathematics 274, 2012.

A. Kurganov and E. Tadmor. Solution of two-dimensional riemann problems for gas dynamics without riemann problem solvers. *Numerical Methods for Partial Differential Equations*, 18, 2002. https://api.semanticscholar.org/CorpusID:10100895.

P. Mocz, M. Vogelsberger, D. Sijacki, R. Pakmor, and L. Hernquist. A discontinuous galerkin method for solving the fluid and magnetohydrodynamic equations in astrophysical simulations. *Monthly Notices of the Royal Astronomical Society*, 437(1):397–414, Oct. 2013. ISSN 1365-2966. doi: 10.1093/mnras/stt1890. http://dx.doi.org/10.1093/mnras/stt1890.

D. Nelson, A. Pillepich, V. Springel, R. Pakmor, R. Weinberger, S. Genel, P. Torrey, M. Vogelsberger, F. Marinacci, and L. Hernquist. First results from the tng50 simulation: galactic outflows driven by supernovae and black hole feedback. *Monthly Notices of the Royal Astronomical Society*, 490(3):3234–3261, Aug. 2019. ISSN 1365-2966. doi: 10.1093/mnras/stz2306. http://dx.doi.org/10.1093/mnras/stz2306.

V. Sacristan. Algorithms for constructing voronoi diagrams, 2019. https://dccg.upc.edu/people/vera/wp-content/uploads/2019/11/GeoC-Voronoi-algorithms.pdf.

D. K. Schaal. Havoc - here's another voronoi code, 2013. Diploma thesis, http://kmschaal.de/Diplomarbeit_KevinSchaal.pdf.

D. K. Schaal. Shocks in the illustris universe and discontinuous galerkin hydrodynamics, 2016. PhD thesis, http://kmschaal.de/PhDThesis_KevinSchaal.pdf.

V. Springel. E pur si muove:galilean-invariant cosmological hydrodynamical simulations on a moving mesh. *Monthly Notices of the Royal Astronomical Society*, 401(2):791–851, Jan. 2010. ISSN 1365-2966. doi: 10.1111/j.1365-2966.2009.15715.x. http://dx.doi.org/10.1111/j.1365-2966.2009.15715.x.

V. Springel, C. Dullemond, P. Girchidis, F. Gräter, R. Pakmor, C. Pfrommer, F. Röke, and R. Klessen. Lecture notes mvcomp1: Fundamentals of simulation methods, 2020.

E. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, 1997.

B. van Leer. *SIAM Journal on Scientific and Statistical Computing*, 5(1), 1984.

B. van Leer. *Communications in Computational Physics*, 1:192, 2006.

W. B. Walter Boscheri, M. D. Michael Dumbser, and E. G. Elena Gaburro. Continuous finite element subgrid basis functions for discontinuous galerkin schemes on unstructured polygonal voronoi meshes. *Communications in Computational Physics*, 32(1):259–298, Jan. 2022. ISSN 1815-2406. doi: 10.4208/cicp.oa-2021-0235. http://dx.doi.org/10.4208/cicp.OA-2021-0235.

C.-M. Zhang, M. G. Knepley, D. A. Yuen, and Y. Shi. Two new approaches in solving the nonlinear shallow water equations for tsunami waves, 2007. https://api.semanticscholar.org/CorpusID:56242963.

Tools

In addition to the mentioned resources, I want to clarify that I used Stackoverflow, C++ Reference multiple Python package documentations and ChatGPT as a C++/Python documentation and for debugging parts of the code.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 22.10.2024