The Local Discontinuous Galerkin Method for the Advection-Diffusion Equation on adaptive meshes

Lukas Dreyer

Born 24th July 1992 in Bergisch Gladbach

22nd February 2021

Master's Thesis Mathematics

Advisor: Prof. Dr. Carsten Burstedde

Second Advisor: Dr. Johannes Holke

Institute for Numerical Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Acknowledgement

First and foremost, I would like to thank my thesis advisors Prof. Dr. Burstedde and Dr. Holke. Thank you for offering your valuable advice and for your understanding in difficult times.

I would also like to thank the experts in numerical weather prediction at JSC for their valuable input: Dr. Griessbach, Dr. Hoffmann, Dr. Meyer and Dr. Stein. Special thanks to my colleagues at the High-Performance Computing departement of the German Aerospace Center for all the inspiring discussions during our digital coffee breaks.

Special thanks also go to my friends Anette, Clelia, David and Johannes for helping me free my mind during our walks through the botanic garden.

The authors gratefully acknowledge the Earth System Modelling Project (ESM) for funding this work by providing computing time on the ESM partition of the supercomputer JUWELS at the Jülich Supercomputing Centre (JSC).

I am extremely grateful to my family for supporting me through my master's degree, for always believing in me and motivating me.

Sina: qatlho' 'ej qamuSHa'qu'

1 Introduction

Computational Fluid Dynamics (CFD) is an important part of numerical simulation, with numerous industrial and scientific applications [2, 9, 14, 35, 58]. A research area with plentiful CFD problems is Numerical Weather Prediction (NWP), see [8, 36–38] and the references therein.

One of the problems of interest is the simulation of the advection and diffusion of a tracer in a given velocity field [34, 47, 61]. An example is the transport of volcanic ash after a volcanic eruption [22, 36, 37].

In this thesis we consider the linear advection-diffusion equation as a model for tracer transport. For a bounded domain $\Omega \subset \mathbb{R}^d$, a time interval $[0, T]$, diffusion coefficient $a \in \mathbb{R}_+$, a velocity field $\vec{c} : \Omega \times [0, T] \to \mathbb{R}^d$ and source and sink function $g : \Omega \times [0, T] \to \mathbb{R}$, we solve

$$u_t + \nabla \cdot (\vec{c}u - a\nabla u) = g \tag{1}$$

for the concentration $u : \Omega \times [0, T] \to \mathbb{R}$ of a tracer with given initial concentration $u_0 : \Omega \to \mathbb{R}$.

Time-dependent partial differential equations (PDEs) like the advection-diffusion equation are often discretized by the method of lines [57]. The problem domain is discretized by a finite mesh. On each of these elements, a small number of values, often called degrees of freedom (DOFs), are used to define a function in a finite dimensional function space on the mesh. The PDE is then solved for the time derivative in this finite dimensional function space. Using a Runge-Kutta timestepping scheme [20, 49], the solution is advanced in time with a well chosen timestep Δt.

The choice of mesh plays a crucial role in the development of numerical schemes [33]. Historically, triangular meshes have been used for their ability to approximate curved meshes [60]. Cartesian meshes on the other hand allow for the simplification of several of the algorithms involved because of their tensor product structure [11].

In numerical simulations, the physical domain Ω is often much larger than the actual region of interest. The simulation of volcanic eruptions on a mesh covering the whole earth is an example of such a simulation with highly localized features. The ash cloud diffuses after being transported for some time, but generally follows the direction of the wind field. For those simulations, using a uniformly refined mesh results in an inefficient use of computing power, since not a lot is happening outside a small area of interest. A potential solution is the use of adaptive mesh refinement (AMR) [12, 13].

There are several approaches to using adaptive meshes in numerical simulations, e.g. unstructured, block-structured and tree based adaptive mesh refinement. Unstructured AMR [52] has the advantage of being conforming, meaning that if two elements share a face, they always share the whole face. A disadvantage is that the adjacency between elements needs to be explicitly

saved. Thus, for dynamic simulations with moving phenomena, a grid generator needs to be used each time the grid is adapted. For block-structured AMR [13, 31], two cartesian grids on different refinement levels are utilized and the solution on the coarser grid is taken as boundary values on the finer grid. Tree-based AMR builds on this approach but allows for refinement of each element individually.

A challenge in exhausting the full computational power of supercomputers is the efficient use of parallel computing. A solution to this challenge is the utilization of tree-based mesh refinement in combination with a space filling curve index to enumerate all refined elements. For parallel load balancing, a linear array is partitioned into equal parts and distributed onto the processors. Especially for moving phenomena with changing meshes this is more efficient than graph partitioning algorithms [15, 19].

In this thesis we investigate the feasibility of simulating transport equations on dynamic adaptive meshes on parallel architectures. We use `t8code` [39, 40] as a grid and data manager. It is a grid and data management library for tree-based adaptive mesh refinement. `t8code` picks up the ideas of `p4est` [19, 41, 42] – a library for quadrilateral and hexahedral grids – and allows for the use of space filling curve indices for different element types.

We discretize the advection-diffusion equation with the local discontinuous Galerkin (LDG) finite element method [25]. It is well suited for parallelization, since the discontinuity of the solution between elements leads to frequent element local operations. The coupling of the solution on adjacent elements is achieved with numerical fluxes on the boundary of elements, an approach that is intensively studied in the finite volume (FV) method [62]. In contrast to the FV method, even higher order methods can be utilized in the LDG method with a small stencil [29]. As a result, communication is only needed with direct face neighbors [4, 59].

We develop a new modular approach for the discretization of the DG methods that is in-line with `t8code`s modular element interface. This will allow using various element shapes such as hypercubes, simplices, prisms, etc. , requiring only modularly adding an implementation of the trial functions and quadrature rules for the elements. We demonstrate the feasibility of our approach by thouroughly investigating the quadrilateral and hexadral case. Based on the results presented in this thesis an extension to other element shapes and hybrid meshes is straightforward.

In addition, we lay the foundations for using the DG methods on meshes with arbitrary curved geometries. In tree-based AMR, the physical domain is meshed with few coarse elements, which are equipped with information about the geometry of the element. We demonstrate how to transfer that information onto each of the refined elements. Numerical tests on a cylindrical mesh with an analytical geometry function show that this approach leads to the same convergence properties as for cartesian meshes.

Furthermore, the modular approach allows us to exchange the numerical

fluxes. We use this flexibility to compare the convergence orders of two common numerical fluxes used in the LDG method.

We adapt the mortar element method [10] to realize the discontinuous Galerkin method on adaptive meshes. We demonstrate the feasibility of this approach by comparing different refinement criteria. With an appropriate refinement criterion, we achieve a significant improvement in the runtime for higher dimensional simulations, while obtaining essentially the same error as for the uniform simulation.

Finally, we make use of the scaling capabilities of `t8code` on parallel architectures. To this end, we extend the computation of the numerical fluxes on the mortars to support adjacent elements on different processes. We investigate the scaling properties of our algorithms and demonstrate near perfect scaling to over 10000 CPU cores on the JUWELS [44] supercomputer in Jülich.

We implement the algorithms presented in this thesis in the `t8dg` [32] application. It is built on C and MPI and will soon be released as free and open source software. We use `t8dg` to simulate the advection and diffusion of a tracer in a cylindrical domain. We plan on extending `t8dg` to the more general set of non-linear Euler equations [7].

This thesis is organized as follows:
In section 2, we summarize the theory of tree based adaptive mesh refinement and space-filling curves.
In section 3, we outline the mathematical principles of the DG method for the linear advection equation. We present some ideas about its implementation and test its validity on uniform meshes.
In section 4, we extend the ideas used in the DG method to the local discontinuous Galerkin method for second order PDEs.
In section 5, we deal with the use of adaptive meshes in the DG method. We introduce the mortar method to calculate boundary fluxes across faces of different levels and discuss interpolation and projection for the refinement and coarsening of the meshes.
In section 6, we finally describe the details to extend our algorithm to MPI based parallel computing.
A summary of our most important results, as well as an outlook on further developements planned for our solver, is found at the end of the thesis in section 7.

The data files which are the basis for the graphs shown in this thesis as well as the `t8dg` source code are enclosed in the electronic medium attached to the thesis.

2 Adaptive mesh refinement

Uniform meshes in a CFD simulation may have billions of elements. We want to use our computational power in parts of the mesh, where a higher resolution leads to the biggest increase in accuracy. This can be achieved by adaptive mesh refinement(AMR).

There are numerous different approaches to AMR, e.g. unstructured refinement, block structured refinement and tree-based refinement.

Unstructured refinement puts the computational stress on the mesh generator. Partitioning of these meshes is done by considering its topology as a graph. Especially for time-dependent simulations, where the features of interest are moving in the computational domain, we often want to dynamically adapt the mesh. At each adapt step, the NP-hard [15] graph partitioning problem has to be solved.

A different approach is to follow the features of interest with a moving patch of higher resolution. The mesh with lower resolution is simulated first and the values on the boundary to the finer mesh are the used as boundary conditions for the simulation on the high resolution grid. In contrast to unstructured mesh refinement, hanging nodes appear between elements of the different patches.

Tree-based mesh refinement combines ideas of both of these approaches. A coarse unstructured grid is used to accurately describe the geometry of the computational domain. Each of these coarse elements is then refined recursively, following a refinement rule for each element type. Thus, the geometric advantages of unstructured mesh refinement are combined with the computational efficiency of structured mesh refinement.

In this chapter, we give a more detailed description of tree-based mesh refinement and its relation to space filling curves. Finally, we describe the grid management software `t8code` and the high level algorithms it provides to change the grid.

2.1 Tree-based mesh refinement

We begin with the definition of a mesh. We consider curved meshes in this thesis, so each element is equipped with a geometry function F_e, mapping a polygonal reference element Ω^e_{ref} in computational space onto the element Ω_e in physical space. We further allow for the meshing of the domain with different element types. We will sometimes omit the element index of the reference domain for brevity.

Definition 1 *Let $\Omega \subset \mathbb{R}^d$ be bounded.*
$\mathcal{T} = \left\{\Omega_e = F_e(\Omega^e_{ref}) | e \in E, F_e : \Omega^e_{ref} \to \Omega_e \text{ diffeomorphism}, \Omega_{ref} \text{ polygon}\right\}$
*is a **legal discretization** of Ω with indices E, if*

(i) $\overline{\Omega} = \bigcup_{e \in E} \Omega_e$,

(ii) *If $\Omega_e \cap \Omega_{e\prime} \neq \emptyset$, then $\Omega_e \cap \Omega_{e\prime}$ is the image under F_e respectively $F_{e\prime}$ of a $(d-k)$-dimensional sub-polygon of the corresponding Ω_{ref} $(0 \leq k \leq d)$.*

We also call a legal discretization a **mesh** and the $(d-1)$-dimensional intersections of elements **faces**.

In tree-based mesh refinement we differentiate between two meshes. The physical domain is meshed with a coarse conforming grid. This is either done by a mesh generator or by hand for simple domains. We recursively refine elements as needed to obtain a fine mesh, which is possibly nonconforming.

Each coarse element of the coarse grid corresponds to the root of a refinement tree. Each element type has a local refinement rule. A typical example is to divide the d-dimensional hypercube into 2^d smaller hypercubes. An example for simplices is the red-refinement. We display these refinement rules in Figure 1.

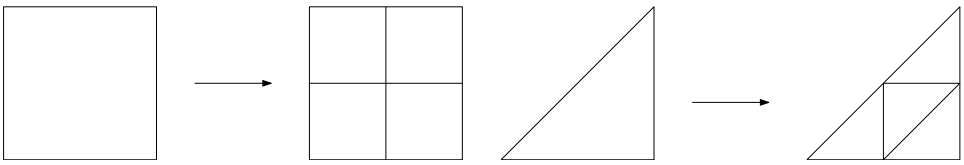

Figure 1: Refinement rules for quadrilaterals and triangles

By defining an ordering of the children in the refinement rule, we can recursively define an ordering on all elements on a certain refinement level. In Figure 2, we display the Z-ordering for quadrilaterals and the resulting ordering of elements in refinement level 2.

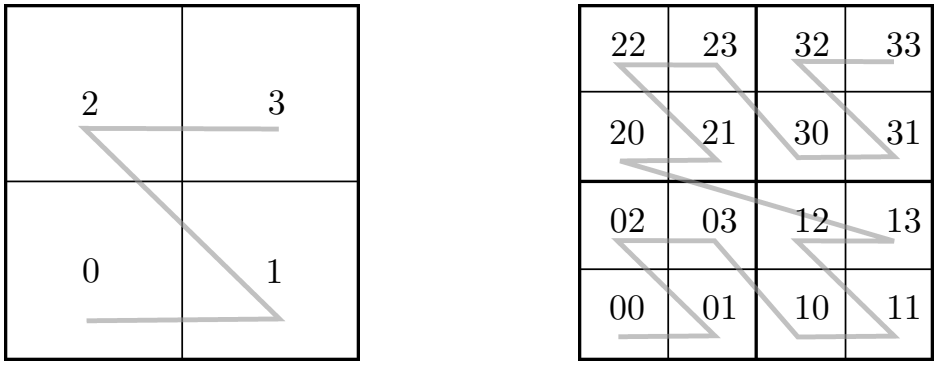

Figure 2: Left: Enumeration of the children in the refinement rule for quadrilaterals. Right: The resulting ordering of all elements on level 2. The pictures are taken from [39] with the authors permission.

The ordering of the children in the refinement process also leads to an enumeration of all elements in an adaptively refined coarse element, see Figure 3. This concept is captured by the idea of space-filling curves.

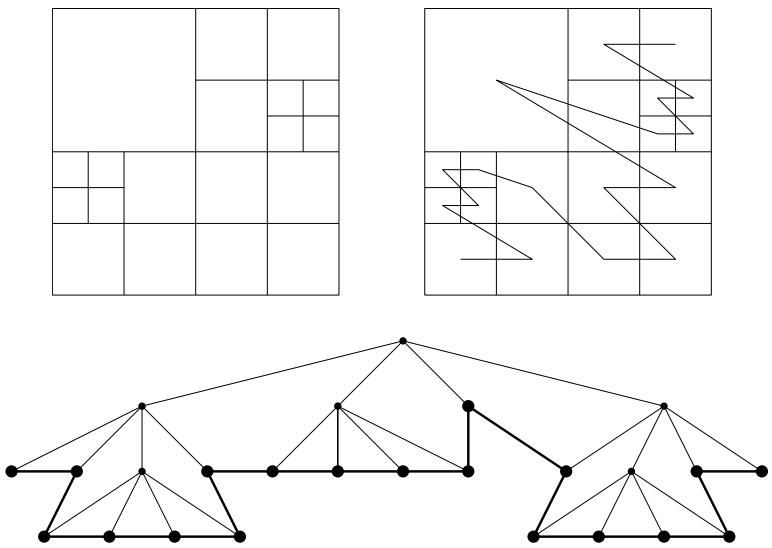

Figure 3: Top left: A square element, which is adaptively refined up to level 3. Top right: The ordering of elements according to the Z-ordering. Bottom: The corresponding ordering in the tree representation of the refinement.

2.2 Space-filling curves

The idea of analytic space-filling curves goes back to Peano. He used them to construct a bijective mapping between the interval $[0, 1]$ and the square domain $[0, 1]^2$ [53]. Many analytic space-filling curves are defined as the limit of discrete space-filling curves on uniformly refined meshes. They have been used in adaptive mesh refinement for a long time [3]. Holke rigorously defines discrete SFCs and its use to enumerate elements in an adaptively refined grid by a so-called SFC-index [39].

The SFC-index arising from the Z-order on hypercubes displayed in Figure 2 is the Morton index. A big advantage is that the index of an element can be efficiently calculated [19]. By using two different orderings for the different orientations of triangles appearing in the red-refinement rule, Holke devised a Morton-type SFC-index for triangles and used this approach to generalize the idea to tetrahedrons [39].

Knapp developed an SFC-index for prisms by using a tensor product of the SFCs for lines and triangles [45]. Knapp also developed an SFC-index for pyramids [46]. This had the additional difficulty, that the typical refinement rule for pyramids divides a pyramid into several smaller pyramids and some tetrahedrons.

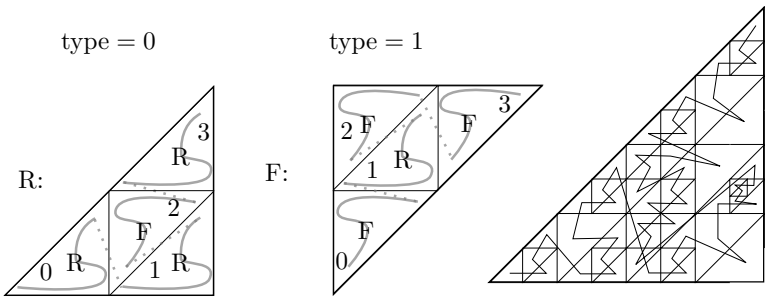

Figure 4: Left and middle: Refinement orders for the two types of triangles occurring in the refinement scheme. Right: Resulting SFC-index for a specific adaptive refinement of a single coarse element. The pictures are taken from [39] with the authors permission.

2.3 Parallel load-balancing

The SFC-index is used to divide the computational load between parallel processes. We divide all elements across all trees into almost equal amounts and distribute them across processes. Each process finds out which processes contain some of its elements after redistribution, and sends these directly via MPI to the new process. Thus, costly All-to-All communication is avoided. There are also some disadvantages compared to graph partitioning. The patch on a process may have bigger surface area and the patches may be disconnected. Burstedde et al. [18] showed that the number of face-connected components of morton-style space-filling curves is bound by 2 in the hypercube case. For triangles and tetrahedra, the bound is dependent on the refinement level l and is $2(l-1)$ for triangles and $2(l+1)$ for tetrahedra.

2.4 The grid management library `t8code`

`t8code`is a library that allows to use different SFCs to manage a tree-based refined mesh. Currently all of the SFCs mentioned above [19, 39, 45, 46] are implemented. Therefore hybrid grids containing cubes and tetrahedrons with prisms and pyramids as connection elements are supported. To add new SFCs to `t8code`, the low-level methods defining the SFC, e. g. parent, child and face-neighbor need to be implemented.

The use of a forest of coarse trees is already implemented in `p4est`, which specializes on hypercubes and uses their tensor structure to further optimize the code. In `t8code`, coarse mesh partitioning is enabled, avoiding the need to save the complete coarse mesh on each process [16]. Currently, for each element, only the construction of its face neighbors is implemented. The construction of vertex neighbours has a lot of further complications and is planned for the future. One of these problems the undetermined number of coarse elements touching at a vertex. As `t8code` uses conforming coarse

meshes, there are only 2 coarse elements touching at a face. In section 3, we will see that the Discontinuous Galerkin methods only needs information about face neighbors of elements.

A typical application uses these high level algorithms provided by `t8code`:

- New
- Adapt
- Balance
- Partition
- Ghost

New constructs a uniformly refined forest from a coarse mesh.
Adapt uses a callback function on every element to indicate whether it needs to be refined or if a family of elements needs to be coarsened.
Balance restores a maximal level difference of 1, by recursively refining elements whose neighbors have a level difference greater than 1.
Partition uses the SFC index to efficiently partition the mesh into blocks of approximately the same number of elements and uses communication between processes to send elements to the correct process.
For each process, **Ghost** determines the processes that contain neighboring elements to the local elements. Through communication, a ghost layer of elements is constructed, so that each process can gain information about the face neighbours of all local elements. As we will later see, a lot of the calculations in this thesis are done independently for each element.

Additionally, for each element we need to construct all of the face-adjacent elements, which can be on a different level than the original element.

3 The Discontinuous Galerkin Method for linear advection

The Discontinuous Galerkin method was first used by Reed and Hill [56] for numerically solving the linear neutron transport equation. Cockburn and Shu extended their approach to non-linear hyperbolic conservation laws in a series of papers [24, 26–28, 30]. They also expanded the approach to second order PDEs with the Local Discontinuous Galerkin method [29] by adapting methods of Bassy and Rebay for the simulation of the Navier-Stokes equation [6]. An overview over the current status of element based galerkin methods and the stabilization techniques used in numerical weather prediction is found in [51].

In this section, we repeat the most important aspects of the DG method, as they are described in [24,26–28,30]. We restrict ourselves to derivation of the scheme for linear advection and mention steps, where special attention needs to be paid in the extension of the scheme to non-linear hyperbolic conservation laws.

First, we introduce the scheme in one dimension, since most of the steps are apparent here. Afterwards, we explain the modifications needed to extend the scheme into higher dimensions. Next, we discuss the choices in implementing the DG method, like the quadrature and functionbasis, the numerical flux and the timestepping method.

The many choices involved in the developement of a DG scheme lead to our new modular approach in implementing the DG method that we describe in subsection 3.9

Finally, we run numerical tests to validate our implementation of the DG scheme for the linear advection.

3.1 Derivation of the scheme

We develop the DG scheme for the scalar linear advection diffusion equation of a tracer in a given velocity field on the unit interval [0,1] with periodic boundary conditions. It reads as follows:

We are given a flow velocity $c \in \mathbb{R}$, an initial concentration $u_0 : [0,1] \to \mathbb{R}$, source and sink terms $g : \mathbb{R} \times [0,T) \to \mathbb{R}$ and want to find the time-dependent pollutant concentration $u : [0,1] \times [0,T) \to \mathbb{R}$, s.t.

$$u_t + c\, u_x = g \qquad on\,(0,1) \times [0,T), \tag{2}$$

with initial conditions u_0 and periodic boundary conditions:

$$\begin{aligned} u(x,0) &= u_0(x) && on\,(0,1) \\ u(0,t) &= u(1,t) && on\,[0,T) \end{aligned}$$

We use a method of lines approach, where we discretize our problem in space and solve the resulting system for the time derivative. This leads to a

system of ordinary differential equations which is then advanced in time by a Runge-Kutta time-stepping method.

For the discretization in space, we use a finite element galerkin approach with discontinuous basis- and testfunctions on a uniform mesh $\mathcal{T}_h$. The Sobolev space H^1 which is normally used for finite element methods is a subspace of the continuous functions. Thus, we extend the function space to square integrable functions L^2, whose restriction on each element is in $\mathcal{P}_k$, the polynomials of degree less than or equal to k.

$$V_h = \{u \in L^2 : u|_{\Omega_e{}^\circ} \in \mathcal{P}_k(\Omega_e{}^\circ) \qquad f.a. \quad \Omega_e \in \mathcal{T}_h\}.$$

The initial condition function u_0 as well as the source function g is interpolated onto on V_h and we solve (2) with solutions in V_h.

$$u(x,t) \simeq u_h(x,t), \quad u_h(\cdot,t) \in V_h \tag{3}$$
$$g(x,t) \simeq g_h(x,t), \quad g_h(\cdot,t) \in V_h \tag{4}$$

As is usual in the treatment of partial differential equations, the strong formulation needs regularity assumptions, and thus does not always define a unique solution. We therefore obtain a weak formulation by multiplying the strong formulation with a testfunction $v \in V_h$ and integrating over each of the elements Ω_e.

$$\int_{\Omega_e} \frac{\partial u_h}{\partial t} v \, dx + \int_{\Omega_e} c \frac{\partial u_h}{\partial x} v \, dx = \int_{\Omega_e} g_h \, v \, dx \tag{5}$$

This weak formulation is element local, since there is no communication between elements. To introduce communication between adjacent elements, we treat the function u_h as smooth and formally apply integration by parts. Since u_h is not uniquely defined on the boundary, we replace it by the numerical flux $(\,)^*$, a trace function on the boundary of elements, which only depends on the value of the flux at both sides of the boundary. Equation (5) then becomes:

$$\int_{\Omega_e} \frac{\partial u_h}{\partial t} v \, dx - \int_{\Omega_e} c u_h \frac{\partial v}{\partial x} \, dx + \int_{\partial\Omega_e} (c u_h)^* \cdot \mathbf{n} \, v \, dS = \int_{\Omega_e} g_h \, v \, dx \tag{6}$$

By rearranging the equation, we get:

$$\int_{\Omega_e} \frac{\partial u_h}{\partial t} v \, dx = \int_{\Omega_e} c u_h \frac{\partial v}{\partial x} \, dx - \int_{\partial\Omega_e} (c u_h)^* \cdot \mathbf{n} \, v \, dS + \int_{\Omega_e} g_h \, v \, dx \tag{7}$$

.

We want to express this weak formulation as a matrix equation, which we can solve numerically. Thus, we write the solution u_h and the source terms

on each element as a linear combination of the element basis functions ϕ_i^e. Here, N_e denotes the number of basisfunctions on element e:

$$u_h|_{\Omega_e{}^\circ} = \sum_{i=0}^{<N_e} \hat{u}_i^e \phi_i^e \tag{8}$$

$$g_h|_{\Omega_e{}^\circ} = \sum_{i=0}^{<N_e} \hat{g}_i^e \phi_i^e \tag{9}$$

We also divide the boundary integral into a sum of the integrals over each of the faces. In the one dimensional case, the faces between elements are zero dimensional vertices. Thus, each function space defined on the faces has dimension one and only one basis function. In preparation for the derivation of the multidimensional scheme, we define a finite dimensional function space on each of the faces f, with N_f basis functions ϕ_i^f.

The numerical flux on a face is then represented as a linear combination of the face basis functions. For each face $\Omega_f \subset \partial\Omega_e$ we get:

$$(cu_h)^* \cdot \mathbf{n}|_{\Omega_f} = \sum_{i=0}^{<N_f} \hat{h}_i^f \phi_i^f \tag{10}$$

Finally, we test (7) against each element basis function, gaining a system of equations for each element. For each test function $v = \phi_j^e$ we get an equation on Ω_e:

$$\begin{aligned} \sum_{i=0}^{<N_e} \frac{d\hat{u}_i^e}{dt} \int_{\Omega_e} \phi_i^e \phi_j^e \, dx = & \sum_{i=0}^{<N_e} c\hat{u}_i^e \int_{\Omega_e} \phi_i^e \frac{\partial \phi_j^e}{\partial x} \, dx \\ & - \sum_{i=0}^{<N_f} \sum_{f \in Faces(e)} \hat{h}_i^f \int_{\Omega_f} \phi_i^f \phi_j^e \, dS \\ & + \sum_{i=0}^{<N_e} \hat{g}_i^e \int_{\Omega_e} \phi_i^e \phi_j^e \, dx \end{aligned} \tag{11}$$

We write the restriction of an element basis function onto a face of the corresponding element as the linear combination of the face basis functions. We collect the coefficients in the matrix Z_f:

Definition 2 *Let e be an element in the mesh and $\Omega_f \subset \partial\Omega_e$ a face of e. Let Φ_e be a functionbasis of Ω_e with cardinality N_e and Φ_f be a functionbasis of Ω_f with cardinality N_f. We define the **face interpolation** matrix $Z^f \in \mathbb{R}^{N_e \times N_f}$ by*

$$\phi_j^e|_{\Omega_f} = \sum_{k=0}^{<N_f} Z_{jk}^f \phi_k^f \tag{12}$$

As a consequence, we write the boundary integral as

$$\int_{\Omega_f} \phi_i^f \phi_j^e \, dS = \sum_{k=0}^{<N_f} Z_{jk}^f \int_{\Omega_f} \phi_i^f \phi_k^f \, dS. \tag{13}$$

In the above formulation, we can extract the element mass and asymmetric stiffness matrices, as well as the face mass matrix.

Definition 3 *Let Ω_e be an element of the mesh with functionbasis Φ^e with N_e basis functions. Each face is equipped with a functionbasis Φ_f with N_f basis functions. Then we define the following matrices:*

*(i) The **element mass** matrix $M^e \in \mathbb{R}^{N_e \times N_e}$ by:*

$$M_{ij}^e = \int_{\Omega_e} \phi_i^e \phi_j^e \, dx \tag{14}$$

*(ii) The **element stiffness** matrix $M^e \in \mathbb{R}^{N_e \times N_e}$ by:*

$$A_{ij}^e = \int_{\Omega_e} \phi_i^e \frac{\partial \phi_j^e}{\partial x} \, dx \tag{15}$$

*(iii) The **face mass** matrix $M^e \in \mathbb{R}^{N_f \times N_f}$ by:*

$$M_{ij}^f = \int_{\Omega_f} \phi_i^f \phi_j^f \, dx \tag{16}$$

We collect the coefficients of the element basis functions of the linear combination of the solution in a vector $\hat{u}^e$. These are commonly called degrees of freedom (dofs). Altogether, we get the following system of equations on each element:

$$M^e \frac{d\hat{u}^e}{dt} = A^e(c\hat{u}) - \sum_{f \in Faces(e)} Z^{f^T} M^f \hat{h}^f + M^e \hat{g}^e, \tag{17}$$

The system is coupled between the elements by the calculation of the numerical fluxes.

Before we further discuss how to calculate these matrix applications, we will see how to extend this scheme to higher dimensions.

3.2 Extension to 2D and 3D

In higher dimensions, the scalar linear advection of a tracer can be stated as follows:
For a given physical domain $\Omega \subset \mathbb{R}^d$ with a divergence free flow velocity

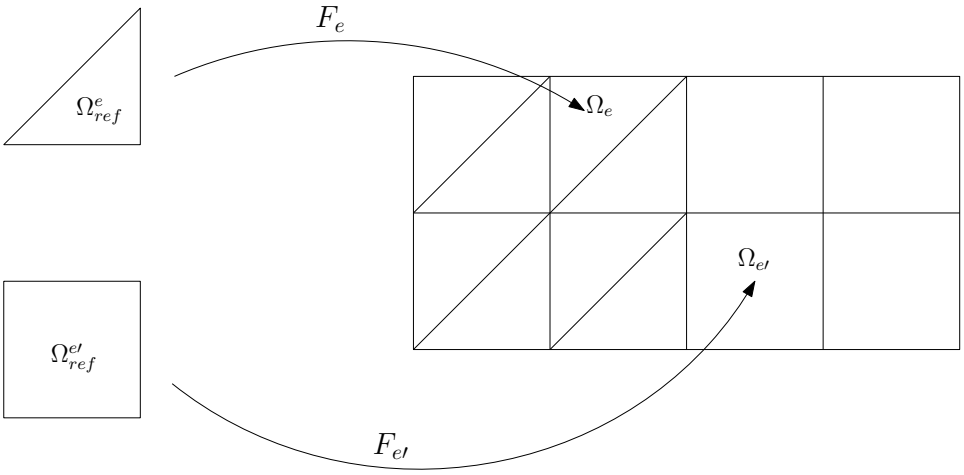

Figure 5: A hybrid mesh consisting of triangles and squares with the corresponding reference elements and geometry functions for two elements in physical space.

field $\vec{c}\colon \Omega \to \mathbb{R}^d$, initial concentration u_0, source and sink terms g, we want to find the time-dependent tracer concentration $u : \Omega \times [0, T) \to \mathbb{R}$, s.t.

$$u_t + \nabla \cdot (\vec{c}u) = g \qquad on\, \Omega \times [0, T). \tag{18}$$

In this thesis we only consider periodic boundary conditions, although in- and outflow boundary conditions are also easily handled by adjusting the calculation of the numerical fluxes [25].

We discretize the physical domain Ω with a uniform mesh $\mathcal{T}_h$. This mesh can contain different element types. We equip each element of the mesh with a geometry function $F_e : \Omega^e_{ref} \to \Omega_e$. It maps a reference element Ω^e_{ref} in computational space onto the element in physical space. In Figure 5, we display the reference elements for a triangle and a square as well as the geometry functions for two elements in physical space.

As in the one dimensional case, we need to define a polynomial trial function space. We obtain the function space on Ω by defining a polynomial function space $\hat{P}(\Omega_{ref})$ on each of the reference elements and transform the basis functions with the element geometry functions into physical space.

$$V_h = \{u \in L^2(\Omega) :\, u|_{\Omega_e \circ} = u^e \circ F_e^{-1} \text{ and } u^e \in \hat{P}(\Omega_{ref}) \qquad f.a. \quad \Omega_e \in \mathcal{T}_h\}$$

For simplices, we consider $\hat{P} = \mathcal{P}_k$, the polynomials of degree less than or equal to k. For hypercubes, we consider $\hat{P} = \mathcal{Q}_k$, the tensor products of one dimensional polynomials of degree at most k.

We approximate u and g in V_h, multiply the strong form of the partial differential equation with a testfunction $v \in V_h$ and integrate over each

element Ω_e to obtain the weak form:

$$\int_{\Omega_e} \frac{\partial u_h}{\partial t} v \, dx + \int_{\Omega_e} \nabla \cdot (\vec{c} u_h) v \, dx = \int_{\Omega_e} g_h \, v \, dx \tag{19}$$

As in the one dimensional case, we assume the solution to be smooth and formally apply integration by parts on the integral with the divergence. We replace the flux on the boundary by a numerical flux $()^*$, since u_h is not defined on the boundary. This results in the following approximation of the integral:

$$\int_{\Omega_e} \nabla \cdot (\vec{c} u_h) v \, dx = -\sum_{k=0}^{<d} \int_{\Omega_e} c_k u_h \frac{\partial v}{\partial x_k} \, dx + \int_{\partial \Omega_e} (\vec{c} u_h)^* \cdot \vec{n} \, v \, dS$$

We replace the approximate solution and the interpolated source function on each element by a linear combination of the element basis functions:

$$u_h|_{\Omega_e{}^\circ} = \sum_{i=0}^{<N_e} \hat{u}_i^e \phi_i^e \tag{20}$$

$$g_h|_{\Omega_e{}^\circ} = \sum_{i=1}^{<N_e} \hat{g}_i^e \phi_i^e \tag{21}$$

We also chose an appropriate functionbasis ϕ^f on each of the faces and write the numerical flux as:

$$(\vec{c} u_h)^* \cdot \vec{n}|_{\Omega_f} = \sum_{i=0}^{<N_f} \hat{h}_i^f \phi_i^f \tag{22}$$

The scheme then reads as follows:

$$\begin{aligned} \sum_{i=0}^{<N_e} \frac{d\hat{u}_i^e}{dt} \int_{\Omega_e} \phi_i^e \phi_j^e \, dx = & \sum_{k=0}^{<d} \sum_{i=0}^{<N_e} c_k \hat{u}_i^e \int_{\Omega_e} \frac{\partial \phi_j^e}{\partial x_k} \phi_i^e \, dx \\ & - \sum_{f \in Faces(e)} \sum_{i=0}^{<N_f} \hat{h}_i^e \int_{\Omega_f} \phi_i^f \phi_j^e \, dS \\ & + \sum_{i=0}^{<N_e} \hat{g}_i^e \int_{\Omega_e} \phi_i^e \phi_j^e \, dx \end{aligned} \tag{23}$$

The main difference compared to the one dimensional case is the appearance of directional stiffness matrices

$$A_{ij}^{k,e} = \int_{\Omega_e} \frac{\partial \phi_j^e}{\partial x_k} \phi_i^e \, dx. \tag{24}$$

3.3 Numerical computation of integrals

In this section, we discuss how to compute the matrix applications involved in the DG method. First, we pull back the integral onto the reference element with the geometry function $F_e : \Omega_{ref} \to \Omega_e$. The basis functions ϕ_i^e are defined as the composition of the basisfunctions on the reference elements with the geometry function for each element. We use the transformation theorem on $\phi_i^e = \phi_i \circ F_e$, where ϕ_i are the basis functions on the reference element:

$$\int_{\Omega_e} \phi_i^e \phi_j^e \, dx = \int_{\Omega_{ref}} \phi_i \phi_j |DF_e| \, dx \tag{25}$$

On the reference element, we approximate the integral with a quadrature rule with Q_e quadrature vertices x_q and quadrature weights w_q:

$$\int_{\Omega_{ref}} \phi_i \phi_j |DF_e| \, dx \simeq \sum_{q=0}^{<Q_e} \phi_i(x_q)\phi_j(x_q)|D_{x_q}F_e|w_q \tag{26}$$

To simplify this representation, we define the Vandermonde matrix, which we precompute once for each element type.

Definition 4 *Let Φ be a functionbasis on a reference element Ω_{ref}^e with N_e basis functions and $\mathcal{Q}$ be a quadrature on Ω_{ref}^e with Q_e quadrature vertices. The **Vandermonde** matrix $V \in \mathbb{R}^{Q_e \times N_e}$ is defined by*

$$V_{qi} = \phi_i(x_q) \tag{27}$$

Here we want to stress that most of the matrices are not stored explicitly. Rather, we implement their application on a vector. Many of the matrix applications involved are sparse. Thus, this approach is more efficient than storing the whole matrix and calculating the matrix vector product each time.

This is especially obvious for the following diagonal weighting matrix, which we use to represent the multiplication with the transformation and the quadrature weight.

Definition 5 *Let Ω_e be an element in the mesh with geometry function $F_e : \Omega_{ref}^e \to \Omega_e$. For a quadrature $\mathcal{Q}$ on the reference element with Q_e quadrature vertices, we define the **element weight** matrix $W \in \mathbb{R}^{Q_e \times Q_e}$ as*

$$W^e = \mathrm{diag}_q(w_q |D_{x_q}F_e|). \tag{28}$$

We are now ready to represent the application of the mass matrix in a short way:

$$M^e = V^T W_e V \tag{29}$$

As a result, we apply the element mass matrix by transforming the vector of degrees of freedom into a vector of values at quadrature points via the Vandermonde matrix. Next, we multiply each value with the corresponding transformation/quadrature weight. Finally we transform the vector of values at quadrature points back into a vector of dofs by applying the transpose of the Vandermonde matrix. We use a similar approach to compute the element stiffness matrix. Here special attention needs to be applied when transforming the gradient. By using the inverse function theorem, we get the following relation between the gradient in physical space $\nabla_x \phi_i^e$ and the gradient in computational space $\nabla_{x_0} \phi_i$

$$\nabla_x \phi_i^e = (D_{x_0} F_e)^{-T} \nabla_{x_0} \phi_i \tag{30}$$

Consequently, we transform the stiffness matrix into physical space in the following way:

$$\begin{aligned}
\int_{\Omega_e} \frac{\partial \phi_j^e}{\partial x} \phi_i^e \, dx &= \left(\int_{\Omega_e} \nabla_x \phi_j^e \, \phi_i^e \, dx \right)_k \\
&= \left(\int_{\Omega_{ref}} D_{x_0} F_e^{-T} \nabla_{x_0} \phi_j \, \phi_i |DF_e| \, dx \right)_k \\
&= \sum_{l=0}^{<d} \int_{\Omega_{ref}} (DF_e^{-T})_{kl} \frac{\partial \phi_j}{\partial x_l} \, \phi_i |DF_e| \, dx
\end{aligned} \tag{31}$$

The integrals in Equation 31 are also approximated by a quadrature rule. We simplify the presentation of Equation 31 by introducing the derivative Vandermonde matrix V^l:

Definition 6 *Let Φ be a functionbasis on a reference element Ω_{ref}^e with N_e basis functions and $\mathcal{Q}$ be a quadrature on Ω_{ref}^e with Q_e quadrature vertices. The **derivative Vandermonde** matrix $V^l \in \mathbb{R}^{Q_e \times N_e}$ is defined by*

$$V_{qi}^l = \frac{\partial \phi_i}{\partial x_l}|_{x_q}. \tag{32}$$

Hence, we can write the the directional stiffness matrix as

$$A^{e,k} = \sum_{l=0}^{<d} (V^l)^T ((DF_e^{-T})_{kl} \, W^e) V, \tag{33}$$

where $(DF_e^{-T})_{kl}$ is evaluated at the corresponding quadrature vertices.

We will further split the calculation of the derivative Vandermonde matrix. To this end, we use the fact that the derivative of a polynomial of degree at most k is again a polynomial of degree at most k. Thus, the derivative of each basis function can be expressed as the linear combination of the basisfunctions in computational space. The coefficents are collected in the derivative matrix.

Definition 7 *Let Φ be a functionbasis on a reference element Ω^e_{ref} with N_e basis functions. We define the* ***directional derivative*** *matrix D^l by:*

$$\frac{\partial \phi_j}{\partial x_l} = \sum_{i=0}^{<N_e} D^l_{ij} \phi_i \tag{34}$$

Therefore, we can evaluate the derivative Vandermonde matrix as the composition of the Vandermonde matrix and the directional derivative matrix D^l, which is only dependent on the chosen functionbasis:

$$\frac{\partial \phi_j}{\partial x_l} | x_q = \sum_{i=0}^{<N_e} D^l_{ij} \, \phi_i(x_q) \tag{35}$$

Thus, we write in short $V^l = D^l V$ and modify the evaluation of the stiffness matrix:

$$A^{e,k} = \sum_{l=0}^{<d} V^T (D^l)^T ((DF_e^{-T})_{kl} \, W^e) V \tag{36}$$

For the calculation of the boundary integrals, we first need to calculate the numerical fluxes $\hat{h}^f$ on each of the faces, then apply the face mass matrix and finally transform the vector of face dofs back into a vector of element dofs. Altogether, we get a system of equations on each element:

$$M^e \frac{d\hat{u}}{dt} = \sum_{k=0}^{<d} A^{k,e} c_k \hat{u}^e - \sum_{f \in Faces(e)} {Z^f}^T M^f \hat{h}^f + M^e \hat{g} \tag{37}$$

To complete the scheme, we need to chose a concrete functionbasis, quadrature, numerical flux, geometry and time-stepping scheme.

3.4 Quadrature and Functionbasis

There are essentially two different approaches that guarantee an easily invertible mass matrix. On the one hand, we can chose a modal basis with orthogonal basis functions. Hence, for each element the mass matrix is diagonal by construction. On the other hand, by chosing a nodal basis where the nodes coincide with the quadrature vertices, the Vandermonde matrix becomes the identity matrix. Thus, the element mass matrix is again diagonal, since it only consists of the transformation and quadrature weights on the diagonal. For our modular implementation of the DG method, we use the second approach and discuss the quadrature rule for hypercubes next.

For the one dimensional case, we chose the Legendre-Gauss-Lobatto (LGL) quadrature rule [1]. This quadrature has the advantage that it includes vertices on the boundary. It has the disadvantage that it does not

exactly compute the integrals appearing in the DG method. As we will see in the numerical tests, this does not appear to have detrimental repercussions for the linear advection equation. For hypercube elements, we use a tensor product quadrature of the one dimensional LGL quadrature.

The quadrature for the faces is obtained by taking the LGL quadrature for elements one dimension lower. We obtain a functionbasis on the faces by using a nodal functionbasis with the face quadrature vertices as nodal basis points. Here, the advantage of the LGL quadrature comes in play. The nodal basis points on each face are a subset of the nodal basis points on each element. Thus, we implement the face interpolation matrix Z^f as a lookup table.

The functionbasis is additionally responsible to supply the derivative matrix. For a nodal basis in one dimension this has been extensively discussed in [48]. We apply the directional derivative matrix in the higher dimensional case by using the tensor product structure. To this end, we apply the one dimensional derivative matrix to the correct subsets of the degrees of freedom.

We plan on extending this approach to quadrature rules for simplical elements in the future.

3.5 Geometry

For each element in the mesh, we need a geometry function mapping the reference element to the image in the physical domain. As we explained in section 2, the geometry of the physical domain is described by a few coarse elements. We equip each coarse element t with a coarse geometry function $F_t^{coarse} : \Omega_{ref}^{coarse} \to \mathbb{R}^d$. We calculate the element geometry function F_e by using information about the position of the refined element e in the coarse reference element Ω_{ref}^t. Each element e has a fine geometry function $F_e^{fine} : \Omega_{ref} \to \Omega_{ref}^{coarse}$, such that the element geometry function is the composition of its fine geometry function and the coarse geometry function of the tree it belongs to:

$$F_e = F_t^{coarse} \circ F_e^{fine} \tag{38}$$

In Figure 6, we show a coarse meshing of a circular disk consisting of 5 coarse quadrilateral elements. One of these coarse elements is once refined. We display the element geometry of one the refined elements as the composition of the coarse and fine geometry function.

For the SFCs currently implemented in `t8code` [17, 39, 45, 46], the fine geometry function consists of scaling, translation and an orthonormal rotation/reflection R:

$$F_e^{fine}(x) = x_0 + hRx \tag{39}$$

In the case of the Morton index for quadrilaterals, we have $h = 2^{-l}$, $R = Id$ and x_0 is the coordinate of the lower left corner of the element [19]. All of

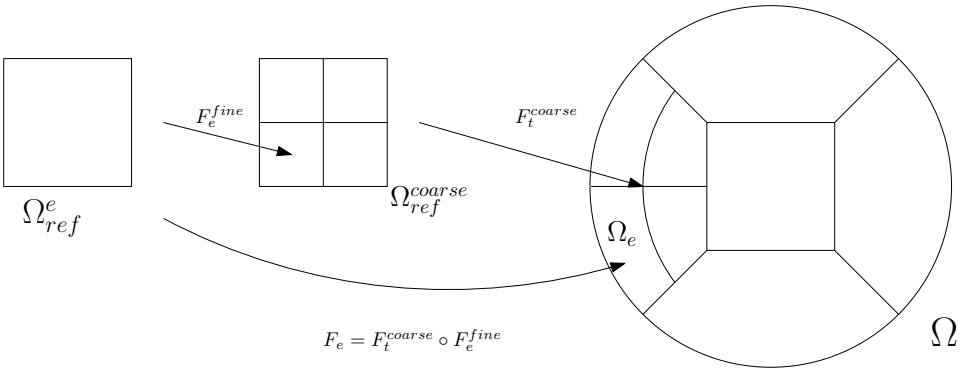

Figure 6: The geometry function is composed of a coarse and a fine geometry function

these values can be queried by `t8code`. Thus, for the Jacobi matrix of the fine geometry function we obtain:

$$DF_e^{fine} = hR \tag{40}$$

For the transformation of the mass matrix, we need the determinant $|D_{x_q} F_e|$ for all quadrature vertices. For the transformation of the gradient and the transformation of the outwards normal, we need $(D_{x_q} F_e)^{-T}$. Thus the coarse geometry also needs to supply the Jacobian of the coarse geometry function. The Jacobian of the element geometry function is determined by the chain rule:

$$D_{x_q} F_e = D_{x_c} F_t^{coarse} \, D_{x_q} F_e^{fine} \tag{41}$$

Here $x_c = F_e^{fine}(x_q)$. After computing the Jacobian of the element geometry function for a quadrature vertex, its determinant and inverse transpose are calculated.

For the transformation of the face mass matrix, we need to take additional care. Let Ω_f be a face of element Ω_e and Ω_{ref}^f be the corresponding face of the reference element Ω_{ref}^e. We obtain the face geometry function by restricting the element geometry function onto Ω_{ref}^f:

$$\begin{aligned} F_f &: \Omega_{ref}^f \to \Omega_f \\ F_f &= F_e|_{\Omega_{ref}^f} \end{aligned} \tag{42}$$

The integrals on the face are transformed by the gram determinant $\sqrt{|DF_f^T DF_f|}$. We obtain the Jacobian of the face geometry function by multiplying it with the set of basis tangential vectors of the face in computational space. For hypercube elements, this is achieved by deleting the column corresponding to the coordinate which is not used in the face. Thus,

by computing the Jacobian of the element geometry function, we can also compute the gram determinant of the face geometry function.

3.6 Numerical flux

The theory of numerical fluxes originates from the finite volume (FV) method, where numerical fluxes are used in the construction of monotone schemes. A numerical flux needs to fulfill the following conditions:

- It only depends on the two values left and right of the interface
- It is locally Lipschitz
- It is consistent with the flux function f, i.e. $(f)^*(a,a) = f(a)$
- It is monotone, i.e. increasing in the first variable and decreasing in the second variable
- It is antisymmetric, i.e. $()^*(a,b) = -()^*(b,a)$

There are several numerical fluxes known in the literature. Unlike in the finite volume method, the discontinuous Galerkin method is less sensitive to the choice of flux because of the higher order polynomials used [25]. In 1D, we chose the upwind flux, as it retains information about the direction of the velocity field:

$$(cu)^*_{\text{upwind}}(u^-, u^+) = \begin{cases} cu^-, c > 0 \\ cu^+, c < 0 \end{cases} \tag{43}$$

For multidimensional problems we use the Lax-Friedrich flux

$$(\vec{c}u)^*_{\text{LF}}(u^-, u^+) = \vec{c}\left(\frac{u^- + u^+}{2}\right) + \frac{C}{2}(u^- - u^+)\vec{n}, \tag{44}$$

where C is an upper bound on the global flow speed. The Lax-Friedrichs flux is easily adaptable to the non-linear case.

In the calculation of the numerical fluxes, we are interested in $()^* \cdot \vec{n}$. Therefore, we can directly calculate:

$$(\vec{c}u)^*_{\text{LF}}(u^-, u^+) \cdot \vec{n} = (\vec{c} \cdot \vec{n})\left(\frac{u^- + u^+}{2}\right) + \frac{C}{2}(u^- - u^+) \tag{45}$$

3.7 Timestepping

The DG method is a discretization in space and needs to be combined with a time-stepping method. Cockburn and Shu used a strong stability preserving explicit Runge-Kutta method in their original papers. It inherits its stability properties from the forward Euler time-stepping method. Combined with

a general slope limiter they were able to prove that their scheme is total variation diminishing. As the focus of this thesis was the implementation of the DG method on adaptive meshes in parallel and good numerical results have been found for other timestepping methods, we chose low memory explicit Runge-Kutta methods [21].

In particular these are Heun's second order method with butcher tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array}, \tag{46}$$

Heun's third order method

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 0 \\ 2/3 & 0 & 2/3 & 0 \\ \hline & 1/4 & 0 & 3/4 \end{array} \tag{47}$$

and the classical Runge-Kutta method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{48}$$

3.8 Stabilization

The numerical approximation of problems with dominating advection may result in unphysical oscillations at shock boundaries. In the case of linear advection, no new shocks are formed, but for non-linear advection, new shocks can be formed. There are several ways to stabilize the scheme and to guarantee its convergence to the entropy solution. In the original papers, generalized slope limiters were used to enforce a total variation diminishing property in the one dimensional case and a maximum principle in the multidimensional case. Since then, plenty different stabilization techniques have been investigated [51]. A technique often used in Numerical weather prediction is artificial diffusion [5]. The motivating application for our thesis is the transport of volcanic ash. It is affected by diffusion, which we consider in section 4. Thus, no additional stabilization is currently implemented. The stabilization of the scheme is an important part of further research.

3.9 Implementation

In this section, we discuss in detail the modular approach we use to implement the discontinuous galerkin method. There are several advantages to this modular approach:

- It can support different element types
- The functionbasis, quadrature, numerical flux and timestepping method can be exchanged
- The modular approach supports further extensions to the solver like the inclusion of a stabilization scheme.

To advance the simulation for a timestep, we need to solve the following system for the time derivate of the vector of degrees of freedom on each element:

$$M^e \frac{d\hat{u}^e}{dt} = \sum_{k=0}^{<d} A^{k,e} c_k \hat{u}^e - \sum_{f \in Faces(e)} {Z^f}^T M^f \hat{h}^f + M^e \hat{g}^e \tag{49}$$

For the advance-step, we need the following matrix applications:

- mass matrix M^e
- inverse mass matrix $(M^e)^{-1}$
- directional asymmetric stiffness matrix $A^{k,e}$
- calculation of the numerical fluxes $\hat{h}_f$
- face mass matrix M^f

We do not want to save these matrices explicitly, but rather calculate their application to a vector. In order to do this efficiently, we can precompute several of the values which we need to apply the matrices. In Figure 7, we depict the structure in which we implemented the calculation of these reused values. Next we describe this structure in detail.

For each element type, we need a quadrature and a functionbasis on the corresponding reference element. In our implementation, the element quadrature and functionbasis also supply the quadrature and functionbasis for each of the faces. We globally precompute the application of the Vandermonde matrix and the directional derivative matrix for each combination of quadrature and functionbasis. Additionally, for hypercubes and prisms, we make use of the tensorproduct form of the element. Thus, if the functionbasis or quadrature adopt the tensorproduct structure, only the lower dimensional Vandermonde and derivative matrices are precomputed. The higher dimensional matrices are applied by applying the lower dimensional matrices on subsets of the vectors.

Additionally, we want to precompute geometry information on each element. We discussed in subsection 3.2, that we need $|D_{x_q} F_e|$ and $(D_{x_q} F_e)^{-T}$ for each quadrature point on the element and $\sqrt{|DF_f^T DF_f|}$ for each quadrature point on the faces. Additionally, for the calculation of the numerical

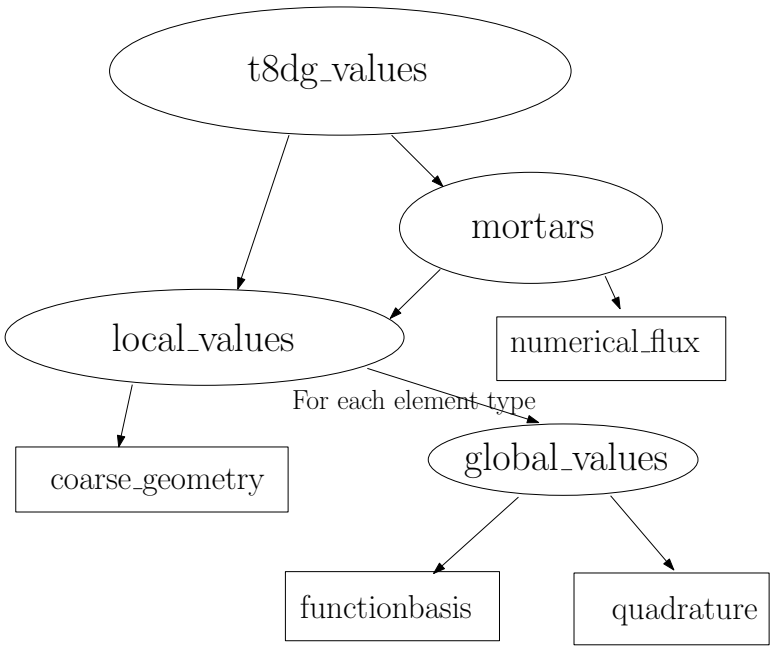

Figure 7: Structure of the `t8dg` implementation: The rectangular boxes represent exchangeable modules, the ellipses represent fixed modules. Each arrow represents that the module from which the arrow originates has a pointer to the module to which the arrow points. In the case of the global values, a module is created for each element type. Thus, the local values module contains an array of pointers to global values.

fluxes, we need the normal vector at the nodal basis vertices of the face function basis. We transform the normal vector for each face of the reference element by $D_x F_e^{-T}$ for each nodal basis vertex x of the face functionbasis.

We utilize mortars for the calculation of the numerical fluxes on the faces. In section 5 we will further discuss their application for non-conforming faces. For conforming faces, we use them to orient the degrees of freedom from both sides of a face.

3.9.1 Mass and Stiffness matrix

The mass, inverse mass and stiffness matrix are completely element local. We save the degrees of freedom of all elements in a linear array in the order of the SFC provided by `t8code`. To apply the respective element matrices, we first need to extract the corresponding element dof values from the array of all dof values. This can be done by using pointer arithmetic. This same idea is also used to inject the calculated values into the result dof vector. A pointer to the element result dof values is given to the function calculating the element matrix application. Thus, the calculated results are directly written into the correct position in the result dof vector.

$$M^e = V^T W^e V$$
$$A^k = \sum_{l=0}^{<d} D^{l^T} V^T ((DF_e)_{kl}^{-T} W^e) V$$

The element mass matrix is implemented in algorithm 1 and the element stiffness matrix is implemented in algorithm 2

Algorithm 1 Element mass matrix M^e

Input: element-dof-values u^e [ ]

Output: element-result-dof-values u^e_{res}[ ]

element-quad-values q^e[ ] = transform-elementdof-to-elementquad(u^e)
q^e = multiply-trafo-quad-weight(q^e)
u^e_{res} = transform-elementquad-to-elementdof(q^e)

The function transform-elementdof-to-elementquad applies the Vandermonde matrix to a vector of element degrees of freedom. The resulting vector of values at quadrature points is multiplied with the precomputed transformation and quadrature weights. By applying the transpose of the Vandermonde matrix, the vector of values at quadrature vertices is transformed back into a dof vector.

Algorithm 2 Element directional stiffness matrix $A^{k,e}$

Input: element-dof-values $u^e[\,]$
Input: directionindex k
Output: result-dof-values[ ]

for all elements e **do**
  $u^e_{res}[\,] = 0$
  for $0 \leq l <$ dim **do**
    element-quad-valuesq^e= transform-elementdof-to-elementquad(u^e)
    q^e = multiply-trafo-quad-weight(q^e)
    q^e = multiply-$(DF_e^{-T})_{kl}(q^e)$
    derivative-values d^e = transform-elementquad-to-elementdof(q^e)
    u^e_{res} += apply-$(D^l)^T(d^e)$
  end for
end for

For the stiffness matrix, we additionally need to multiply each vector of values at quadrature points with $(D_{x_q}F_e^{-T})_{kl}$ and apply the transpose of the directional stiffness matrix $(D^l)^T$ to the dof vector.

3.9.2 Numerical flux

First, we need to calculate the numerical flux on the face degrees of freedom. We demonstrate this in algorithm 3. Afterwards, we apply the face mass matrix. This algorithms is essentially the same as algorithm 1, although the weighting matrix consists of the face quadrature weights multiplied with the gram determinant of the face geometry function. We explained in subsection 3.5 how we precompute these values for each element. Finally, we transform the degrees of freedom on the face back to degrees of freedom on the element with the transpose of the face interpolation matrix. The face interpolation matrix is implemented as a lookup table for the functionbasis that we use.

3.10 Theoretical results

In 1974, Raviart and Lesaint [50] first investigated the original DG method by Reed and Hill for the linear neutron transport [56]. They proved kth order convergence in space in the L^2-norm for general triangulations and $(k+1)st$ order convergence for tensor products of polynomials of degree k. Johnson and Pitkaränta [43] improved the result for general triangulations

Algorithm 3 Numerical flux

Input: dof-values[ ]
Input: numerical-flux()
Output: face-numerical-flux-values [ ]

for all elements e **do**
    u^e[ ] = extract-element-dof-values (dof-values,e)
    for all f ∈ Faces(e) **do**
        e_f = face-neighbor-element
        u^{e_f}[ ] = extract-element-dof-values (dof-values,e_f)
        $u^{e,f}$[ ] = extract-face-dof(u^e,f)
        $u^{e_f,f}$[ ] = extract-face-dof(u^{e_f}, f)
        face-numerical-flux-values[ ] = numerical-flux($u^{e,f}$,$u^{e_f,f}$)
    end for
end for

to order $(k + \frac{1}{2})$, which was numerically shown to be optimal [54]. All of these results assume that the solution is smooth.

Cockburn et al. analyzed in [24] the spatial convergence of the Runge-Kutta Discontinuous Galerkin method for non-linear advection. They showed that if the quadrature rule over the faces is exact for polynomials of degree $2k+1$ and the quadrature rule over the elements is exact for polynomials of degree $2k$, the DG method obtains a $k+1$ order convergence in space. Zhang and Shu extended that analysis to the fully discrete problem for scalar conservation laws [63] and symmetrizable conservation laws [64] with a second order total variation diminishing Runge Kutta method. In these papers, they also describe that numerical tests show the necessity of the exactness of the quadrature rule for non-linear conservation laws.

The LGL quadrature rule with $k + 1$ integration points is only accurate for polynomials of degree up to $2k - 1$ [55]. Thus, for the non-linear case, we do not expect the optimal order of convergence. For the linear case that we implemented in this thesis, we will investigate the order of convergence in the following tests.

3.11 Numerical tests

We conduct numerical tests to validate our implementation of the DG method for linear advection. First, we investigate the order of convergence in time and in space. We measure the error in the relative L^2 norm. To evaluate this norm, we use the LGL quadrature rule that we also use in the DG

scheme. This is not exact for the polynomials of degree $2k$ that we need to evaluate in the L^2 norm. However, it has the advantage that we can use the element mass matrix from subsubsection 3.9.1 on the squared DOF vector to evaluate the element L^2 norm.

As a first test, we analyze the convergence on the unit interval $[0, 1]$ with periodic boundary conditions. The line need one coarse element to model its geometry, with the identity function as its coarse geometry function.

As is usual for finite element methods, the error of the discontinuous galerkin method depends on the regularity of the solution. Thus, for the first round of tests we take as an initial function the following cosine function on the unit interval $[0, 1]$:

$$f(x) = \cos(2\pi x). \tag{50}$$

For the velocity field we use the constant velocity field $c(x) = 1$. For T=1, the velocity field has transported the initial function once around the periodic interval. Thus, the analytical solution equals the initial function.

In `t8dg`we implemented linear, quadratic and cubic polynomials in one dimension and their tensor product function spaces in higher dimensions. We combine these with the second, third and fourth order explicit time stepping methods discussed in subsection 3.7.

Each of the graphs for the three following numerical tests shows the L^2 error for a fixed Runge-Kutta method combined with every implemented function space.

To investigate the convergence in space, we fix the timestep $\Delta t = 0.001$ and vary the uniform refinement level from $l = 2$ to $l = 10$. In Figure 8, we plot the L^2 error against the refinement level. Each diagram shows the L^2 error for a fixed time-stepping order and all tested polynomial degrees. In the first diagram, we see that the error decreases with the level until the size of the error in time dominates. When we chose the level too high, the CFL-condition is no longer fulfilled. As a result, the scheme is no longer stable and doesn't converge anymore.

For the third order time-stepping scheme, we observe a smaller error in time, which leads to an improvement in the error for higher levels. Still, for polynomials of degree 3, we recognize a small region where the error in time dominates before the scheme becomes unstable. For the fourth order time-stepping scheme, we can identify the order of convergence in space for each of the polynomial orders. We quantify the order of convergence by calculating the ratio of the errors for the last two levels, for which the scheme was stable with the Runge-Kutta scheme of order 4. For polynomials of degree 2, the error decreases by a factor of 3.958, for polynomials of degree 3, the error decreases by a factor of 7.968 and for polynomials of degree 4, the error decreases by a factor of 15.995. This means that we indeed achieve convergence in space of order $(k + 1)$ for polynomials of degree k, even with the inexact quadrature that we use.

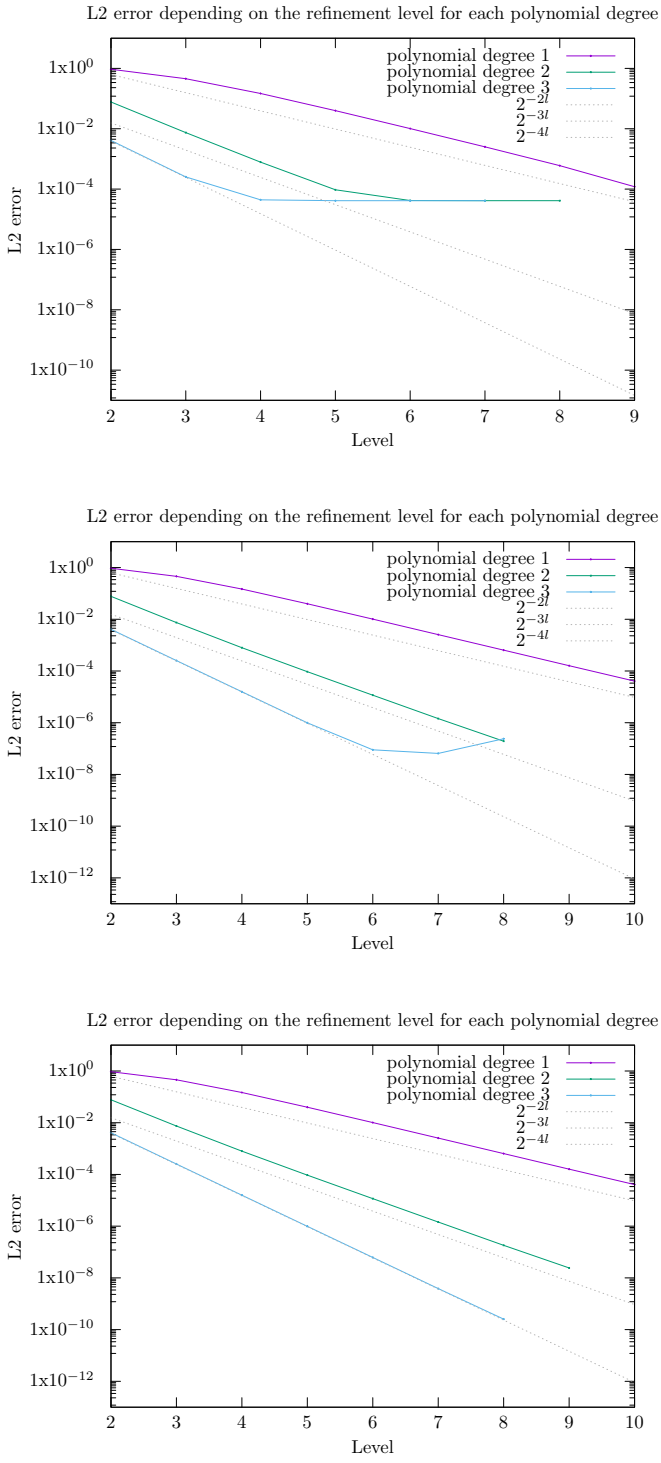

Figure 8: L^2-error for a fixed timestep $\Delta t = 0.001$ depending on the refinement level. Top: Runge-Kutta order 2, Middle: Runge-Kutta order 3, Bottom: Runge-Kutta order 4.

Next we examine the convergence in time. We fix the level at $l = 5$ and increase the number of timesteps by a factor of 2 for each run, hence decreasing the timestep Δt by a factor of 2. For explicit timestepping schemes, the timestep needs to be large enough to guarantee stability. For the discontinuous galerkin methods, this is ensured by a CFL condition. The missing datapoints in Figure 9 belong to simulations in which the timestep was too big. As a consequence, the method did not converge. There is then a small region in which decreasing the timestep results in a higher accuracy but then the error in space dominates. As a result, we conclude that on the one hand, it is important to chose the timestep small enough to ensure convergence. On the other hand, further decreasing the timestep only leads to an improvement in the error for a small region, after which the error in space dominates. For the fourth order Runge-Kutta method, further decreasing the timestep does not lead to any improvements in the error.

Finally, we test which polynomial order should be combined with which timestepping order. To this end, we fix the sufficiently small CFL=0.05 and determine the timestep Δt by

$$\Delta t = CFL \cdot \Delta x. \tag{51}$$

As before, we combine each polynomial degree with each timestepping scheme. The results are shown in Figure 10. It is evident that polynomial basis functions of degree k should be combined with a Runge-Kutta scheme of order $k + 1$, since using a higher order time-stepping scheme does not improve the error. With this approach, we obtain the optimal $(k + 1)st$ order of convergence.

For this reason, we conduct all further test with polynomials of degree k and timestepping methods of order $k + 1$. We extend the tests to the multidimensional case. We take as an initial function the following cosine product on the unit hypercube $[0, 1]^d$:

$$u_0(\vec{x}) = \prod_{i=1}^{d} \cos(2\pi x_i) \tag{52}$$

For the velocity field we use the constant velocity field in x direction, i. e. $\vec{c}(x) = \begin{pmatrix}1\end{pmatrix}$ in 1D, $\vec{c}(x) = \begin{pmatrix}1\\0\end{pmatrix}$ in 2D and $\vec{c}(x) = \begin{pmatrix}1\\0\\0\end{pmatrix}$ in 3D.

We show the results in Figure 11. For the one dimensional simulation with polynomials of degree 3, we reach a resolution in which machine precision leads to a deterioration of the error. Up to the refinement level 10, we still achieve fourth order convergence. In all other cases, we obtain the optimal $(k + 1)st$ order of convergence.

Finally, we verify our approach for uniform meshes with a curved geometry. We consider a circle ring in 2D and a cylinder ring in 3D. We mesh

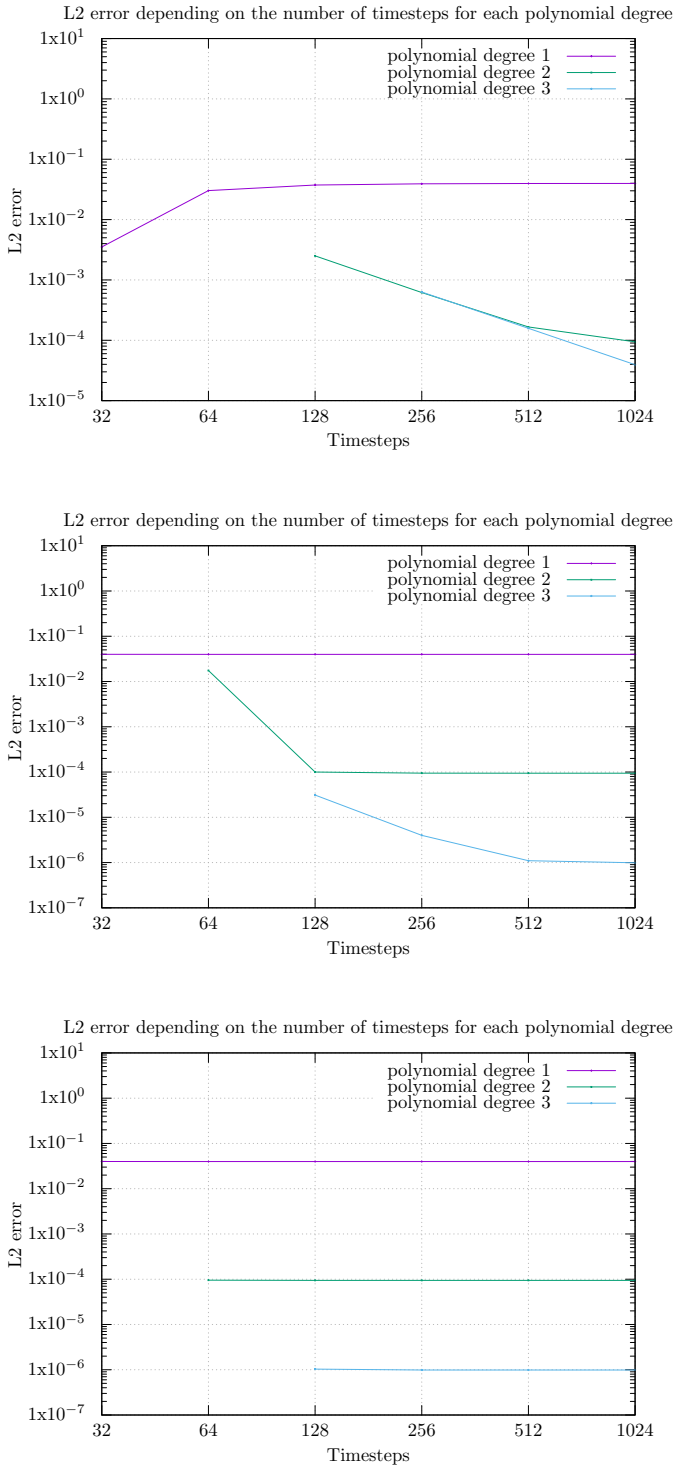

Figure 9: L^2-error for a fixed level $l = 5$ depending on the number of timesteps. Top: Runge-Kutta order 2, Middle: Runge-Kutta order 3, Bottom: Runge-Kutta order 4.

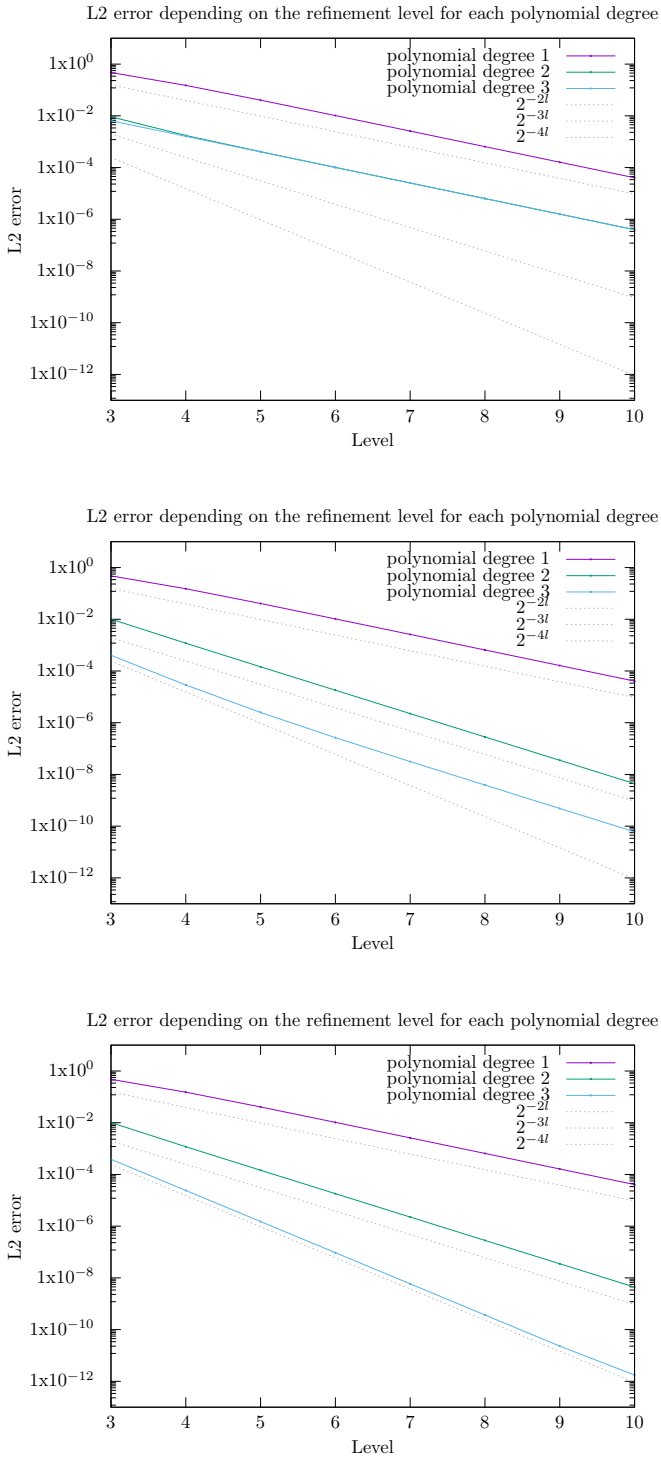

Figure 10: L^2-error for a fixed CFL-number $CFL = 0.05$ depending on the refinement level. Top: Runge-Kutta order 2, Middle: Runge-Kutta order 3, Bottom: Runge-Kutta order 4.

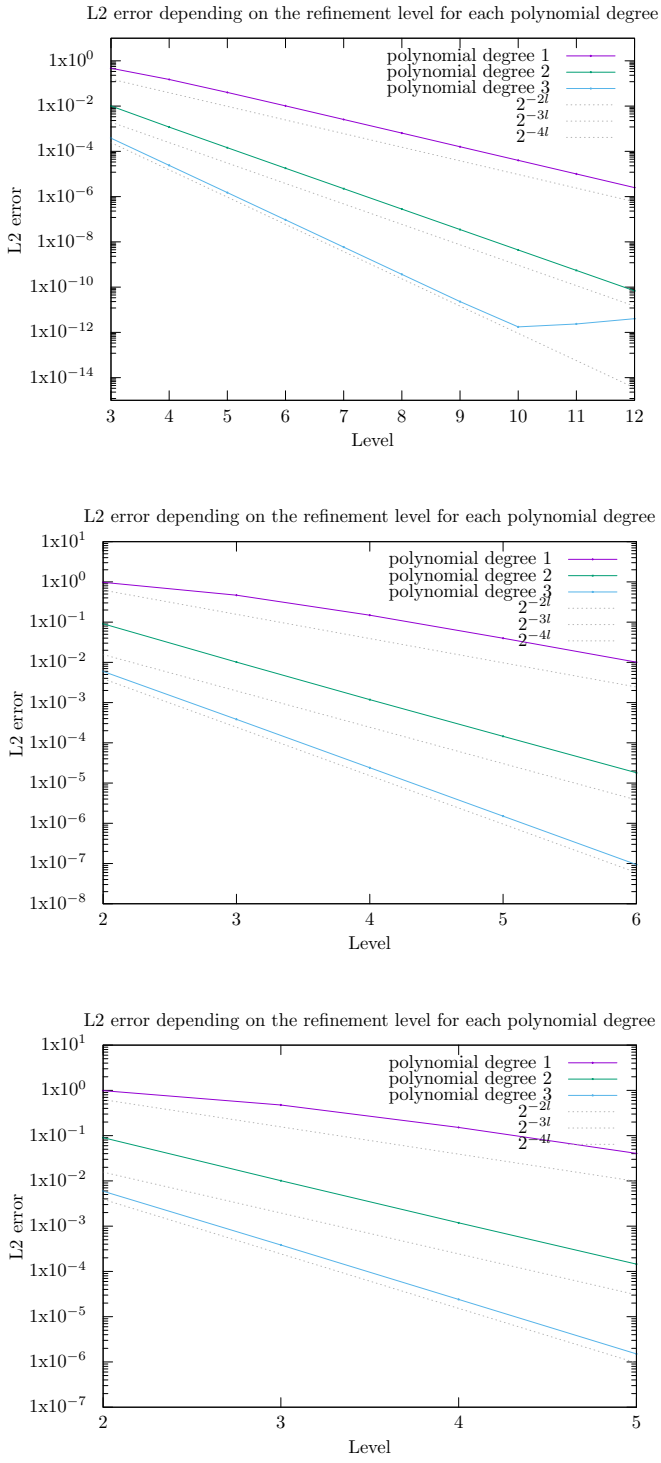

Figure 11: L^2-error for polynomials of degree k combined with Runge-Kutta methods of order $k+1$ depending on the refinement level. Top: 1D, Middle: 2D, Bottom: 3D.

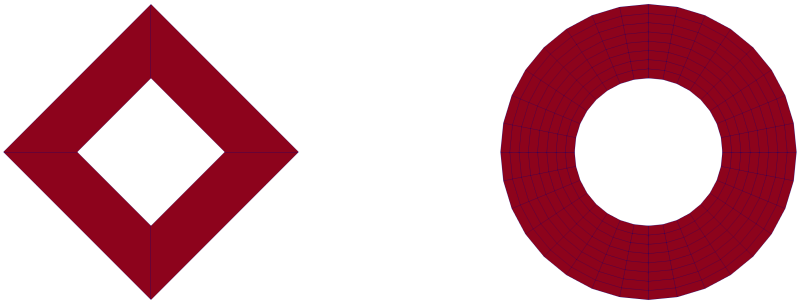

Figure 12: Left: The coarse mesh of the circle ring. It is equipped with a geometry, but we currently only use linear geometry when displaying the elements. Right: The uniformly refined mesh at level $l = 3$.

the circle ring with 4 trees and the following coarse geometry functions:

$$\begin{aligned} F_t : [0,1]^2 &\to \Omega^t_{coarse} \\ F_t \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} &= \begin{pmatrix} r(\hat{x})cos(\theta_t(\hat{y})) \\ r(\hat{x})sin(\theta_t(\hat{y})) \end{pmatrix} \end{aligned} \tag{53}$$

Here, $r(\hat{x}) = r_i + (r_o - r_i)\hat{x}$ and $\theta_t(\hat{y}) = (t + \hat{y})\frac{2\pi}{4}$, with inner radius $r_i = 1$, outer radius $r_o = 2$ and tree index $t \in \{0, 1, 2, 3\}$.

In Figure 12, we display the coarse mesh and a refined mesh of level $l = 3$. Although the coarse mesh is already equipped with the geometry, we currently don't use the capabilities of paraview to display curved elements. After refining, each fine element uses the geometry function of the coarse element. Because of the higher resolution, the output of the fine mesh approximates the geometry of the circle ring.

For the initial function, we use

$$u_0 \begin{pmatrix} x \\ y \end{pmatrix} = \sin\left(\arctan\frac{y}{x}\right) \sin\left(2\pi\left(\sqrt{x^2 + y^2} - 1.5\right)\right). \tag{54}$$

We also use the rotational velocity field

$$\vec{c} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x \end{pmatrix}. \tag{55}$$

After a full rotation, the initial function is rotated once around the circle ring, thus the analytical solution equals the initial function.

We display the achieved L^2 error for refinement levels $l = 2, \ldots, 7$ in Figure 13. We check the order of convergence by calculating the ratio of the

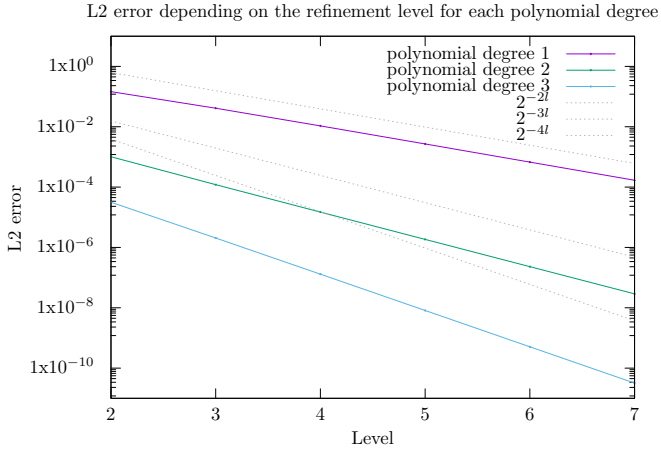

Figure 13: L^2-error for the simulation of the advection on a circle ring geometry in 2D depending on the refinement level.

errors for refinement level 6 and 7. We obtain an improvement in the error by a factor of 3.997 for linear polynomials, by a factor of 8.001 for quadratic polynomials and by a factor of 15.992 for cubic polynomials. This shows that our approach for analytic curved geometries does not deteriorate the convergence order of the scheme.

Next, we consider the 3-dimensional case of the cylinder ring. We mesh the cylinder ring with 4 trees, which are connected in the z-direction and the following coarse geometry functions:

$$F_t : [0,1]^3 \to \Omega^t_{coarse}$$
$$F_t \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{pmatrix} = \begin{pmatrix} r(\hat{x})cos(\theta_t(\hat{y})) \\ r(\hat{x})sin(\theta_t(\hat{y})) \\ \hat{z} \end{pmatrix} \tag{56}$$

Here, $r(\hat{x}) = r_i + (r_o - r_i)\hat{x}$ and $\theta_t(\hat{y}) = (t + \hat{y})\frac{2\pi}{4}$, with inner radius $r_i = 1$, outer radius $r_o = 2$ and tree index $t \in \{0, 1, 2, 3\}$.

For the initial function, we use

$$u_0 \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sin\left(\arctan\frac{y}{x}\right) \sin\left(2\pi\left(\sqrt{x^2+y^2} - 1.5\right)\right) \sin\left(z - 0.5\right). \tag{57}$$

We use the spiraling velocity field

$$\vec{c} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} y \\ -x \\ 1 \end{pmatrix}. \tag{58}$$

We display the achieved L^2 error for refinement levels $l = 2, \ldots, 7$ in Figure 14. We check the order of convergence by calculating the ratio of the

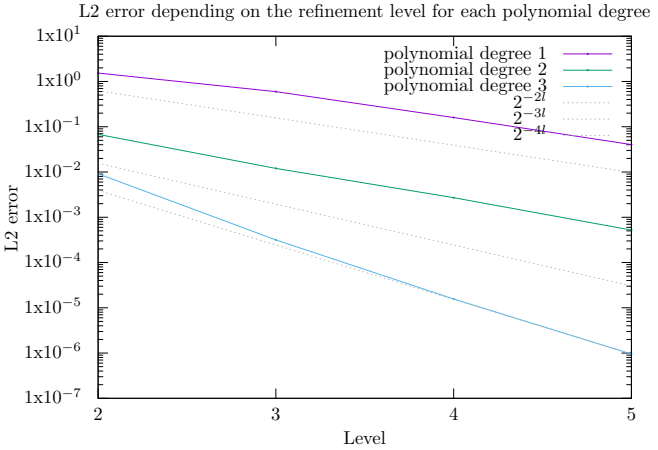

Figure 14: L^2-error for the simulation of the advection on a cylinder ring geometry in 3D depending on the refinement level.

errors for refinement level 4 and 5. We obtain an improvement in the error by a factor of 3.954 for linear polynomials, by a factor of 5.203 for quadratic polynomials and by a factor of 16.421 for cubic polynomials. Thus, we achieve optimal orders of convergence for linear and cubic polynomials. We obtain a sub optimal order of convergence for quadratic polynomials. Based on the other results in this section, we think that for this case the region of optimal convergence is not yet reached. We will further investigate this case with simulations on a higher level in the future.

For solutions with lower regularity, we expect that using a higher order method does not have the same order of convergence as for smooth functions. Thus, we test the scheme in one dimension with different orders on a function which is not continuously differentiable. We use as initial function $u_0(x) = 1-|0.5-x|$, with the constant velocity field $c(x) = 1$ up to T=1. In Figure 15 we display the results and see that we only have first order convergence independent of the order of the method.

In `t8dg`we don't yet use slope limiters, which ensure lower order convergence in the appearance of shocks in the solution. In the linear advection case, no shocks form on their own. Already apparent shocks lead to non physical Gibbs oscillation anyway. To demonstrate this, we use as initial function $u_0(x) = \mathbb{I}_{[a,b]}$ and a refinement level $l = 9$. We display the initial function and the resulting oscillations for linear, quadratic and cubic polynomials at $T = 1$ in Figure 16. Increasing the order of the method decreases the amplitude of the oscillations, but increases the frequency.

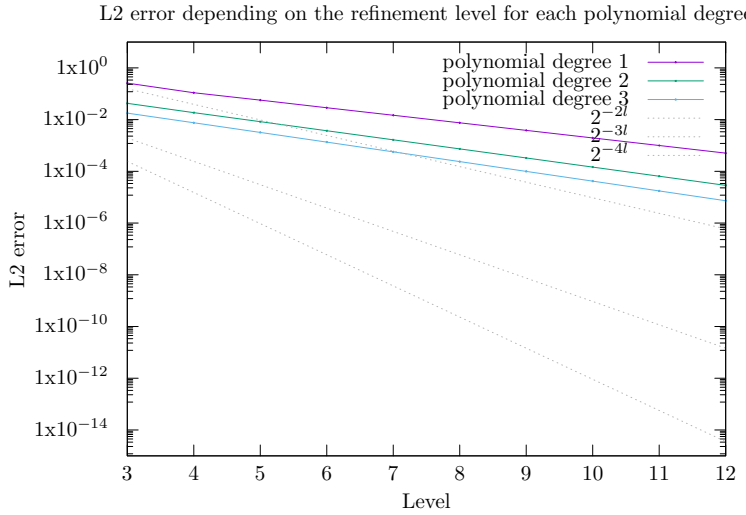

Figure 15: L^2-error for the simulation of the advection of the non-smooth hat function in 1D, depending on the refinement level.

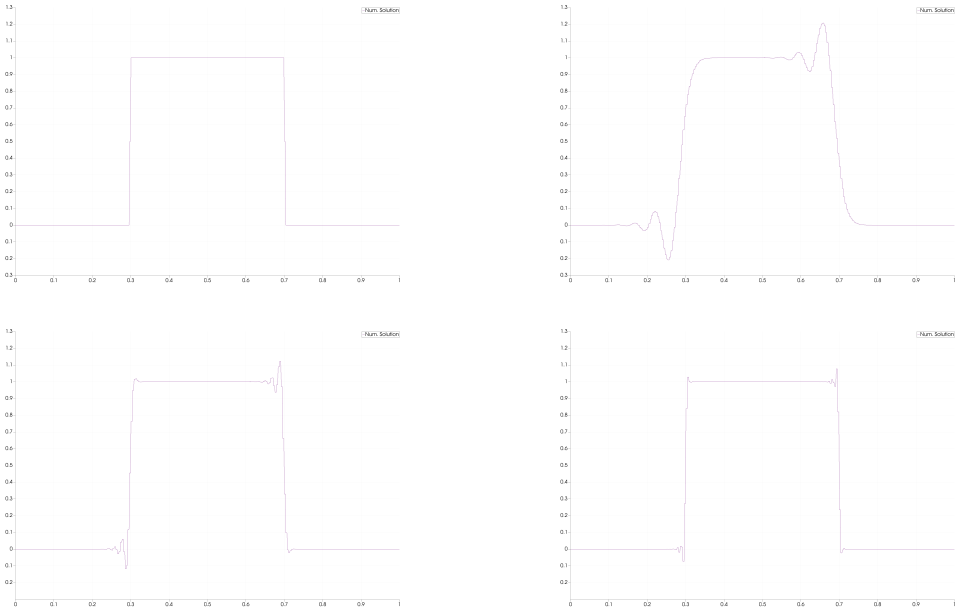

Figure 16: Non physical Gibbs oscillation appear at shocks. Top left: The initial function u_0. The other pictures display the numerical solution at T=1 for different polynomial degrees. Top right: linear polynomials. Bottom left: quadratic polynomials. Bottom right: cubic polynomials.

Summary

- We develop a modular approach to solve the Discontinuous Galerkin method for the linear advection for arbitrary element types on curved geometries and implement the quadrature and functionbasis for hypercubes.
- The scheme shows optimal $(k+1)st$ order of convergence for polynomials of degree k combined with explicit Runge-Kutta timestepping methods of order $k+1$
- Higher order methods are only useful if the solution is smooth enough.
- To extend the method to nonlinear hyperbolic equations, we need to consider generalized slope limiters.

4 The Local Discontinuous Galerkin method for diffusion

In this chapter, we develop the local discontinuous galerkin method for the advection-diffusion equation. The derivation closely follows [29].

We consider the following formulation of the advection-diffusion of a tracer. On a physical domain $\Omega \subset \mathbb{R}^d$ with given divergence free flow velocity field $\vec{c} : \Omega \to \mathbb{R}^d$, diffusion coefficient $a \in \mathbb{R}_+$, initial concentration of the tracer $u_0 : \Omega \to \mathbb{R}$ and source and sink terms $g : \Omega \times [0, T) \to \mathbb{R}$, we want to find the time-dependent concentration of the tracer $u : \Omega \times [0, T) \to \mathbb{R}$, s.t.

$$u_t + \nabla \cdot (\vec{c}u - a\nabla u) = g \qquad in\, \Omega \times (0, T) \tag{59}$$

with periodic boundary conditions.

In the original paper, the diffusion coefficient is chosen as a symmetric, positive semidefinite matrix. We only consider $a \in \mathbb{R}_+$ for clarity and refer to [29] for the handling of the more general case.

In section 3 we presented the DG method for linear hyperbolic conservation laws. Its principles can also be used to extend the method to nonlinear conservation laws. An intuitive approach for second order PDEs like the advection diffusion equation would be to use the gradient operator as the non linear flux function. This unfortunately leads to a constant error for the heat equation [29]. The alternative is to introduce additional variables representing the gradient. Therefore, we need to solve a system of hyperbolic conservation laws.

4.1 Derivation of the scheme

First, we note that since $a \in \mathbb{R}_+$, we can take the square root $b = \sqrt{a}$. We then define new scalar variables $q_k = b\frac{\partial u}{\partial x_k}$. Thus, we can rewrite the problem (59) as follows:

$$u_t + \sum_{k=0}^{<d} \frac{\partial}{\partial x_k} (c_k u - b q_k) = g \tag{60}$$

$$q_k - b\frac{\partial u}{\partial x_k} = 0 \tag{61}$$

We chose the same finite dimensional function space V_h as in the case of the DG method for hyperbolic conservation laws for both the concentration and gradient variables. We obtain the following weak formulation of the system of equations:

$$\int_{\Omega_e} u_t v dx + \sum_{k=0}^{<d} \int_{\Omega_e} \frac{\partial}{\partial x_k} \left(c_k u - b q_k\right) v dx \qquad = \int_{\Omega_e} g v dx \tag{62}$$

$$\int_{\Omega_e} q_k w dx - \int_{\Omega_e} b \frac{\partial u}{\partial x_k} w dx \qquad = 0 \tag{63}$$

As in section 3, we formally apply integration by parts and manage the discontinuity of u and q_k on the element boundaries by introducing a numerical flux $()^*$:

$$\int_{\Omega_e} u_t v dx + \int_{\partial\Omega_e} \left(\vec{c} u - b\vec{q}\right)^* \vec{n}\, v\, dS \tag{64}$$

$$- \sum_{k=0}^{<d} \int_{\Omega_e} \left(c_k u - b q_k\right) \frac{\partial v}{\partial x_k} dx \qquad = \int_{\Omega_e} g v dx \tag{65}$$

$$\int_{\Omega_e} q_k w dx - \int_{\partial\Omega_e} \left(bu\right)^* n_k\, w dS + \int_{\Omega_e} bu \frac{\partial w}{\partial x_k} dx \qquad = 0 \tag{66}$$

where n_k denotes the k^{th} component of the outside normal vector $\vec{n}$.

We replace u, q_k and g by their approximation in the finite dimensional function space V_h and represent them in the corresponding basis. Additionally, we represent the numerical fluxes in the basis of the faces:

$$u|^{\mathrm{o}}_{\Omega_e} \simeq \sum_{i=0}^{<N_e} \hat{u}_i^e \phi_i^e \tag{67}$$

$$q_k|^{\mathrm{o}}_{\Omega_e} \simeq \sum_{i=0}^{<N_e} \hat{q_k}_i^e \phi_i^e \tag{68}$$

$$g|^{\mathrm{o}}_{\Omega_e} \simeq \sum_{i=0}^{<N_e} \hat{g}_i^e \phi_i^e \tag{69}$$

$$\left(\vec{c} u - b\vec{q}\right)^* \cdot \vec{n}|_{\Omega_f} \simeq \sum_{i=0}^{<N_f} \hat{h}_{u,i}^f \phi_i^f \tag{70}$$

$$\left(bu\right)^* n_k|_{\Omega_f} \simeq \sum_{i=0}^{<N_e} \hat{h}_{q_k,i}^f \phi_i^f \tag{71}$$

Now, we can extract the same matrix applications as in section 3.

$$M^e \frac{d\hat{u}^e}{dt} = \sum_{k=0}^{<d} A^{k,e} \left(c_k \hat{u}^e - b\hat{q}_k^e\right) - \sum_{f \in Faces(e)} {Z^f}^T M^f \hat{h}_u^f + M^e \hat{g} \tag{72}$$

$$M^e \hat{q}_k^e = \sum_{f \in Faces(e)} {Z^f}^T M^f \hat{h}_{q_k}^f + A^{k,e}(b\hat{u}^e) \tag{73}$$

We already discussed in detail the implementation of matrix implementations in our modular approach in subsection 3.9. Hence, we only need to chose the numerical fluxes to complete the description of the LDG method.

4.2 Numerical flux

The numerical fluxes should only depend on the interior and exterior values as well as the normal vector $\vec{n}$.

- locally Lipschitz
- conservative
- consistent
- allow for a local resolution of q_k in terms of u only.
- reduce to an E-flux, when the diffusion coefficient $a = 0$
- enforce L2-stability

We split the numerical flux into an advective and a diffusive part:

$$(\vec{c}u - b\vec{q})^* \left(u^-, u^+, \vec{q}^-, \vec{q}^+\right) = (\vec{c}u)^*_{adv} \left(u^-, u^+\right) - (b\vec{q})^*_{diff} \left(\vec{q}^-, \vec{q}^+\right) \tag{74}$$

$$(bu)^*(u^-, u^+, \vec{q}^-, \vec{q}^+) = (bu)^*_{diff}(u^-, u^+) \tag{75}$$

For the advective part, we use the same fluxes as in section 3, i. e. the upwind flux in 1D and the Lax-Friedrichs-flux in higher dimensions. For the diffusive part, we compare the central fluxes

$$(b\vec{q})^*_{central} \left(q^-, q^+\right) = b\, \frac{\vec{q}^- + \vec{q}^+}{2}, \tag{76}$$

$$(bu)^*_{central}(u^-, u^+) = b\, \frac{u^- + u^+}{2} \tag{77}$$

and the alternating fluxes:

$$(b\vec{q})^*_{alternating} \left(q^-, q^+\right) = b\, \vec{q}^- \tag{78}$$

$$(bu)^*_{alternating}(u^-, u^+) = b\, u^+ \tag{79}$$

We could also consider $\vec{q}^+$ and u^- for the alternating fluxes, as long as for each face we take the numerical flux for the concentration and the gradient from different sides of the face.

4.3 Theoretical results

Bassi and Rebay [6] introduced the idea of splitting the second order PDEs into a system of first order PDEs which could be solved with the Discontinuous Galerkin method. They used the central flux and reported an order of convergence of $(k+1)$ for polynomials with an even degree k, and an order of convergence of k for polynomials of odd degree.

Cockburn and Shu [29] proved this results and extended the analysis to a more general set of numerical fluxes. For this general class of fluxes, they proved a convergence order of k for polynomials of degree k. However, they reported an order of convergence of $k+1$ for the alternating fluxes, a special instance of this general class of numerical fluxes. This was later proved by Castillo [23].

4.4 Numerical tests

To test convergence, we need an initial function for which we know the exact solution. Thus, we consider the cosine product on the hyperube $[0,1]^d$:

$$u_0(\vec{x}) = \prod_{i=1}^{d} \cos(2\pi x_i) \tag{80}$$

For the problem without advection, it has the following analytical solution:

$$u(\vec{x},t) = e^{-ad(2\pi)^2 t} \prod_{i=1}^{d} \cos(2\pi x_i) \tag{81}$$

We run tests both for the advection-diffusion equation and for the case of pure diffusion combined with both the central and the alternating flux. For each simulation, we use polynomials of degree k together with time-stepping schemes of order $k+1$, where $k \in \{1,2,3\}$.

For the advection-diffusion, we use a constant velocity field in x-direction with magnitude 1. Thus, at T=1 the analytical solution equals the analytical solution in the case of pure diffusion.

We chose a time-step for the advection-diffusion equation with a CFL condition with a sufficiently small constant CFL number $CFL = 0.05$. For the case of pure diffusion, the CFL condition is more restrictive. We chose a CFL number proportional to $\sqrt{\Delta x}$ to ensure stability.

In Figure 17, we show the results of the simulation of the advection-diffusion equation with diffusion coefficient $a = 0.0001$, with the alternating

flux. We run the tests for d-dimensional hypercubes with $d \in \{1, 2, 3\}$. We achieve optimal $(k+1)st$ order in all space dimensions.

We repeat the tests for the advection-diffusion equation with the central flux and show the results in Figure 18. The convergence order is the same as for the alternating flux. The difference in the error between the alternating and the central flux is not distinguishable from the graphics, but the error achieved with the central flux is consistently about 10% higher than the error achieved with the alternating flux.

Next, we test our algorithms on the purely diffusive case. The results for the alternating flux are shown in Figure 19 and the results for central flux are shown in Figure 20. We again achieve $(k+1)st$ order of convergence for alternating flux. In contrast, for the central flux, we only achieve third order convergence for polynomials of degree 3. This is in line with the theoretical results, which only guarantee k^{th} order convergence for polynomials of odd degree k for the central flux. For linear polynomials, we achieve second order convergence, which is better than the theoretically expected first order convergence.

Summary

- We introduced the LDG method by splitting the advection-diffusion equation into a system of first order PDEs for the gradient and the solution variable.
- We locally solve the equations for the gradient with the ideas from the DG method. The gradient values are then used to compute the time derivative.
- We tested the implementation with the central and alternating flux for the diffusive numerical flux and achieved optimal order of convergence for the advection-dominated simulation for both fluxes.
- For the purely diffusive simulation, the alternating flux achieved optimal order of convergence for all polynomial degrees. The central flux achieved sub optimal, but expected third order convergence for polynomials of degree 3.

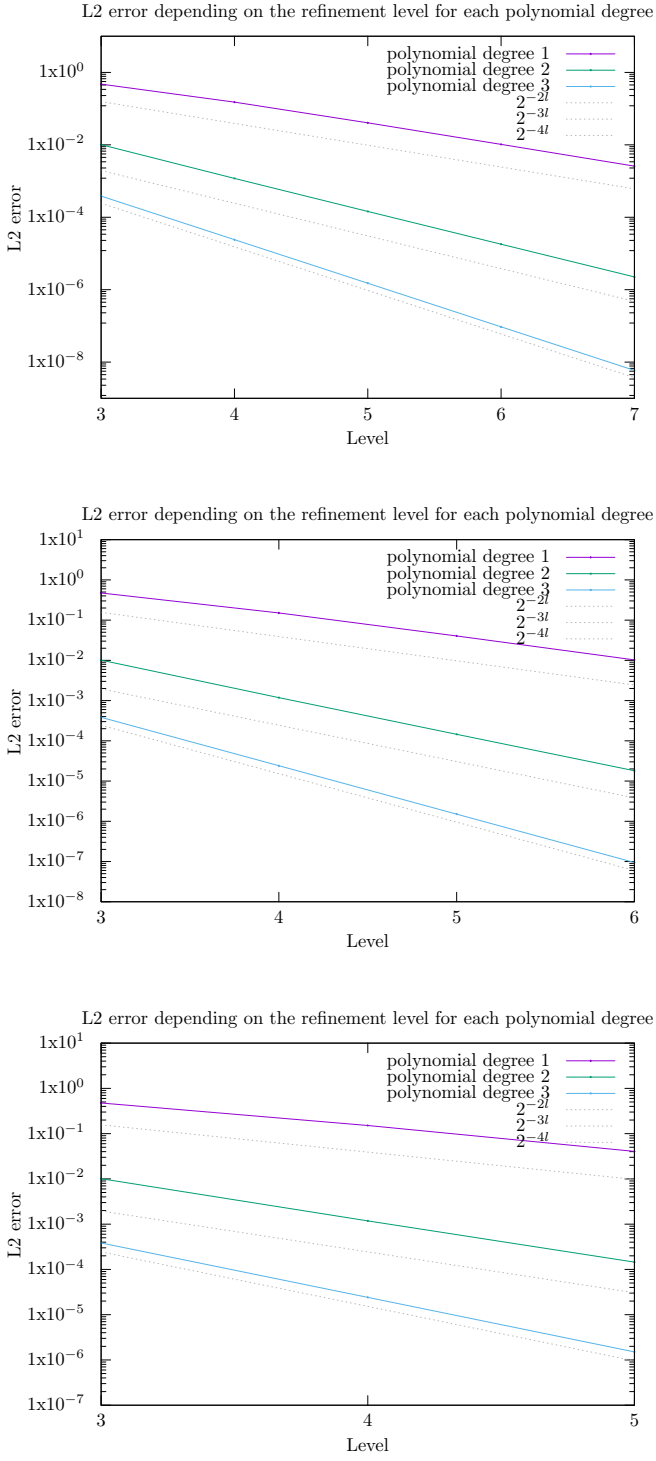

Figure 17: L^2-error for polynomials of degree k combined with Runge-Kutta methods of order $k+1$ depending on the refinement level for the advection-diffusion equation with the alternating flux. Top: 1D, Middle: 2D, Bottom: 3D.

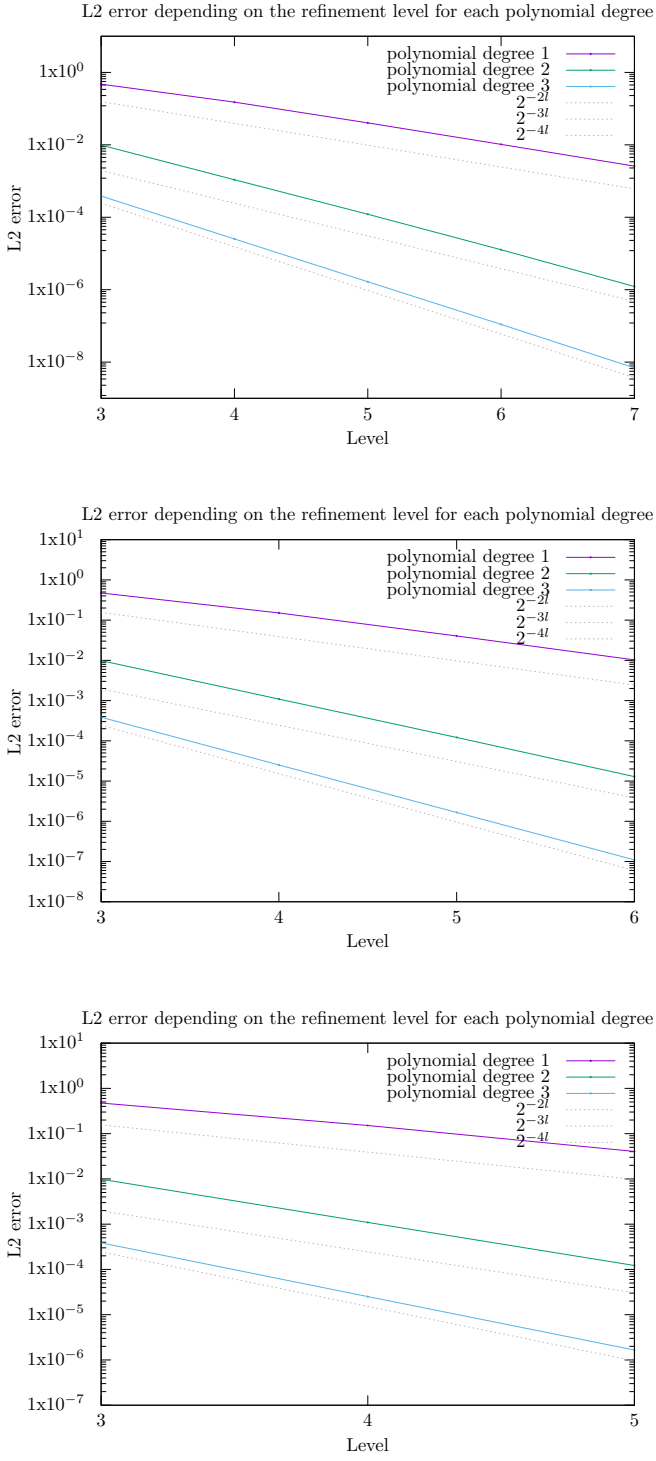

Figure 18: L^2-error for polynomials of degree k combined with Runge-Kutta methods of order $k+1$ depending on the refinement level for the advection-diffusion equation with the alternating flux. Top: 1D, Middle: 2D, Bottom: 3D.

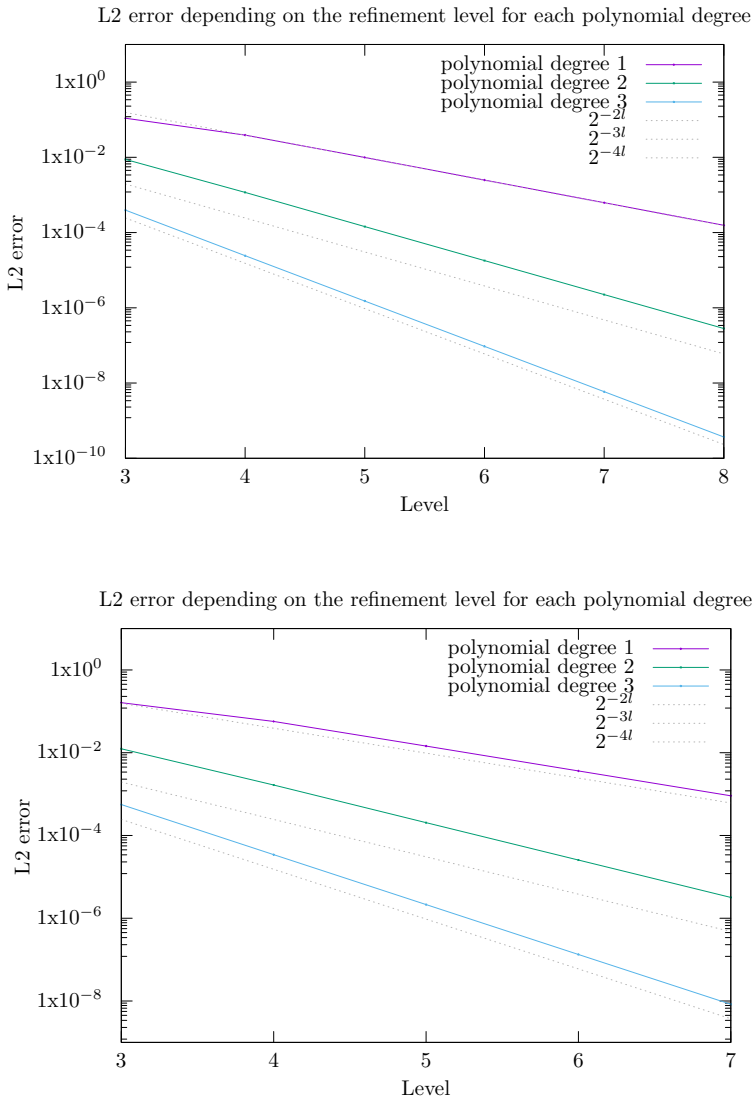

Figure 19: L^2-error for polynomials of degree k combined with Runge-Kutta methods of order $k+1$ depending on the refinement level for the advection-diffusion equation with the alternating flux. Top: 1D, Middle: 2D, Bottom: 3D.

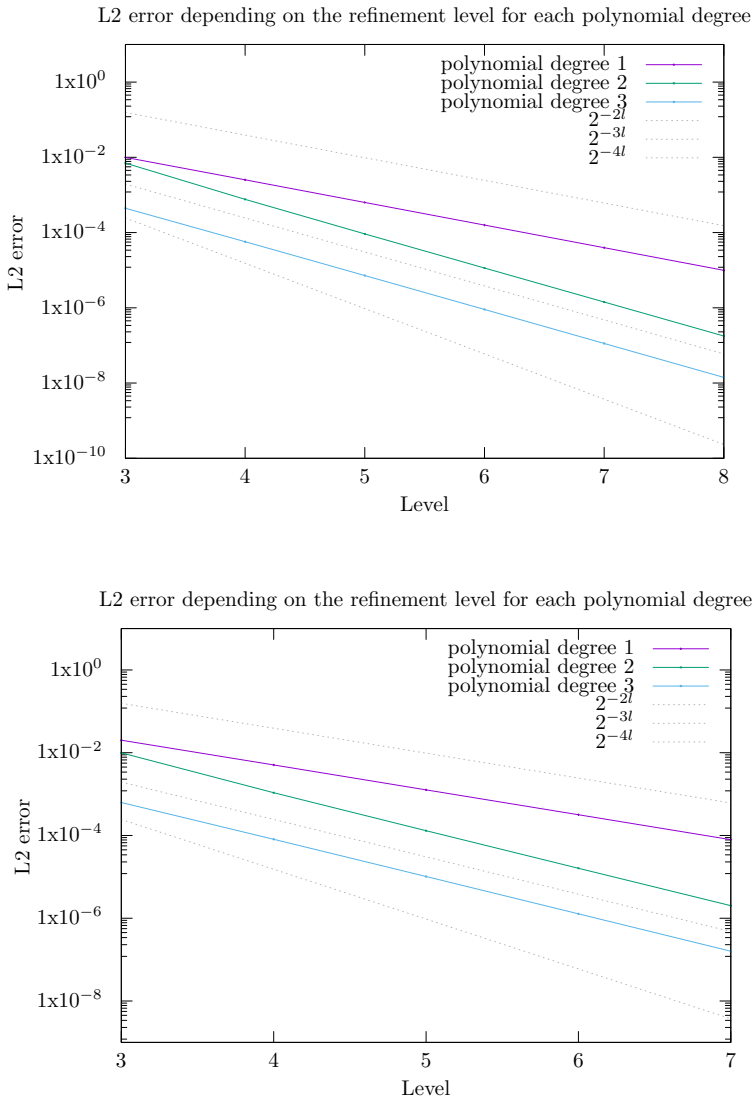

Figure 20: L^2-error for polynomials of degree k combined with Runge-Kutta methods of order $k+1$ depending on the refinement level for the advection-diffusion equation with the alternating flux. Top: 1D, Middle: 2D, Bottom: 3D.

5 The Discontinuous Galerkin method on adaptive meshes

In this chapter we describe the implementation of the DG method using adaptive grids based on the DG method on uniform grids described in section 3. Therefore we do not repeat structures and principles but concentrate on the adjustments we make to handle adaptive grids. As explained in section 2, we use the grid managing software `t8code`in order to handle adaptive meshes. We first explain how to decide which elements need to be refined and which families of elements should be coarsened.

Next, we describe how to interpolate and project the degrees of freedom on a changing mesh. In the end, we discuss the changes to the calculation of the numerical fluxes, which is the only change we need to do in every iteration of calculating the time derivative, since all other operations are element local and don't need any information about other elements.

5.1 Refinement and Coarsening of a mesh with `t8code`

To adapt a mesh, `t8code` utilizes a user defined callback function. This callback function has as input either a single element or a family of sibling elements. For a single element, the callback function has to decide whether the element should be refined or not. For a family of elements, the callback function has three options. Either the first element is refined, all sibling elements are coarsened together, or nothing should happen right now. In the last case, the callback function is called on all single elements except the first, since maybe some of these should be refined.

We describe the different refinement criteria that we use in the next section.

To simplify the handling of nonconforming faces, we enforce a maximal level difference of 1 between face-adjacent elements. In `t8code`, this ability is provided by the high-level algorithm balance. It is separated from the adapt procedure. Thus, the mesh is first coarsened and refined according to the adapt criterion. Afterwards, the mesh is balanced by repeatedly refining elements that are more than one level coarser than at least one of their face-adjacent elements. This can lead to a ripple effect [39].

Only after calling the balance routine, we compute the local geometry information of the new elements and interpolate and project the degrees of freedom. We discuss this in detail in subsection 5.3

5.2 Adapt criteria

The refinement criterion is an important part of every adaptive scheme. Ideally, an a posteriori error estimator is used to refine the mesh in areas where the error is big.

For the simulation of tracer transports, a simple refinement criterion is a mass density criterion with thresholds, since the grid should be refined in areas where the tracer appears in the first place.

For this criterion, we calculate the average density over an element and refine an element if this value is above a given refinement threshold. We coarsen a family of elements, if the average density of all elements is below a given coarsening threshold. Throughout this thesis, we use a refinement threshold of 0.1 and a coarsening threshold of 0.05.

A different approach is to refine the mesh in areas where the gradient is big. A common criterion that is scaling invariant is the relative min-max criterion. On each element, we calculate the difference of the maximal dof value and the minimal dof value and divide it by the minimal dof value. The resulting value is compared with refinement a refinement threshold of 0.1 and a coarsening threshold of 0.01. To circumvent problems in regions of little to no density of the tracer, we combine the relative min-max criterion with a mass threshold.

To compare the performance of the mass and relative min-max criterion, we also introduce an artificial refinement criterion especially for the initial function that we investigate in the numerical test. For this function, the interesting features are located in a moving circle ring with inner radius r_i, outer radius r_o and center $\vec{x}_c(t)$. Thus, we chose a threshold $\delta = 0.01$ and refine an element if the distance r between the element midpoint and the circle center fulfills $r_o - \delta < r < r_i + \delta$. We coarsen a family of elements if the distance r lies outside that interval for every element in the family.

In Figure 21, we show the initial function that we use in the numerical tests on a mesh that is refined according to the artificial refinement criterion. The mesh is also balanced, otherwise the families of elements in the corners and in the middle of the square would be coarsened once more.

5.3 Interpolation and Projection

We compute the interpolation of the degrees of freedom on the reference elements. Here, we can precompute the interpolation matrix P^r for each functionbasis of the child element r. With this matrix we can represent $u^r = P^r u^e$, where u^r is the vector of degrees of freedom of the child element and u^e is the vector of degrees of freedom of the parent element.

For each functionbasis on the line, we precompute the interpolation matrix for both children. To apply the interpolation in higher dimensions, we apply the one-dimensional interpolation matrix on the correct subsets of the element dofs.

For the projection in the coarsening process, we use L^2 projection. Thus, we want to minimize the L^2 difference between the projected function u^e and the functions u^r defined on the children elements:

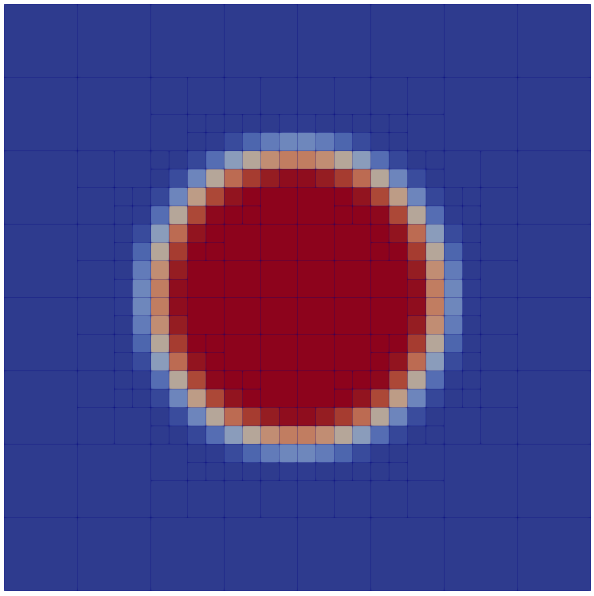

Figure 21: Refinement of a quadrilateral mesh around the boundary of a circle. We enforce a 2:1 balance. Thus, the elements in the middle of the circle and in the corners of the square are not coarsened more.

$$\min_{u^e}\left(J(u^e)\right) = \min_{u^e}\left(\sum_{r\in children(e)}\int_{\Omega_r}\frac{1}{2}\left(\sum_j u_j^e\phi_j^e - \sum_j u_j^r\phi_j^r\right)^2 dx\right) \tag{82}$$

We obtain the minimum by finding extremal points of the functional:

$$0 = \frac{\partial J}{\partial u_i^e} = \sum_{r\in children(e)}\int_{\Omega_r}\left(\sum_{j=0}^{<N_e} u_j^e\phi_j^e - \sum_{j=0}^{<N_r} u_j^r\phi_j^r\right)\phi_i^e dx \tag{83}$$

$$= \sum_{j=0}^{<N_e} u_j^e\int_{\Omega_e}\phi_i^e\phi_j^e dx - \sum_{r\in children(e)}\sum_{j=0}^{<N_r} u_j^r\int_{\Omega_r}\phi_i^e\phi_j^r dx \tag{84}$$

Thus, by rearranging and extracting the mass matrix M^e and the mixed mass matrices $M_{ij}^{r,e} = \int_{\Omega_r}\phi_i^e\phi_j^r dx$, we get

$$M^e u^e = \sum_{r\in children(e)} M^{r,e}u^r. \tag{85}$$

We already need the element mass matrix M^e for the calculation of the time derivative. The mixed mass matrices can be calculated by interpolating each basis function on the element e to the basis functions on the children elements:

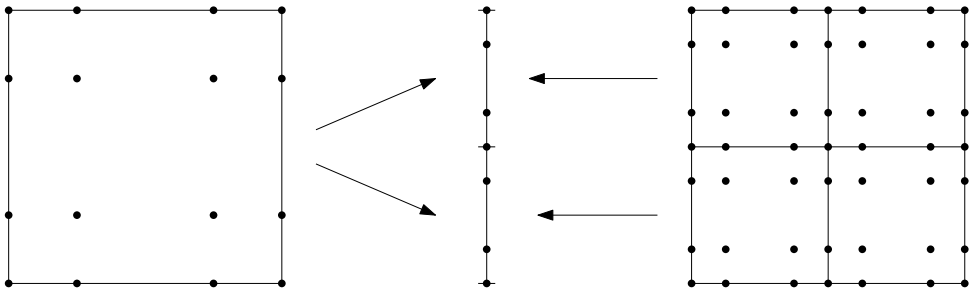

Figure 22: Face mortar between two elements of differing levels with 4 LGL basis nodes on the faces.

$$\int_{\Omega_r} \phi_i^e \phi_j^r dx = \sum_{k=0}^{<N_r} P_{ik}^r \int_{\Omega_r} \phi_k^r \phi_j^r dx \tag{86}$$

Thus $M_e^r = P^{rT} M^r$, where M^r is the actual mass matrix of the child element r.

Combining the above equations results in:

$$u^e = (M^e)^{-1} \sum_{r \in children(e)} P^{rT} M^r \tag{87}$$

5.4 Calculation of numerical fluxes

To complete the description of the adaptive scheme, we describe how to compute the numerical fluxes on hanging nodes. For two elements on differing levels with a common face, we want to calculate the flux through the face.

To this end, we use mortars on the faces. They have the advantage of being flexible to further modifications to our solver, like implementing p-adaptivity [51] and supporting different element types on each side. In Figure 22, we show the face mortar between two quadrilaterals with differing refinement levels.

There are two common approaches in the literature, pointwise matching or using an L^2-projection. We decided to use the L^2-projection, since it helps enforcing the conservation of the solution variable [51]. Additionally, we already need to calculate the L^2-projection for the coarsening of elements and we can reuse subroutines for the L^2-projection on the faces.

To calculate the numerical flux, we first interpolate the dofs on the bigger face to the smaller subfaces. We then apply the numerical fluxes on each of the subfaces. In the end, we use the L^2-projection back onto the bigger face. See the algorithm in algorithm 4. In `t8dg`, we only create the mortars for each face once, so that we don't calculate the numerical fluxes twice.

Algorithm 4 Mortar Numerical Fluxes

Input: element e
Input: face f
Input: dof-values u [ ]
Input: numerical-flux()
Output: numerical flux DOF vector for element e on face f
e^f = faceneighbours(e,f)
if level(e)==level(e^f) **then**
  u^- = transform-elementdof-to-facedof(u^e,f)
  u^+ = transform-elementdof-to-facedof(u^{e^f},f)
  flux = numerical-flux(u^-, u^+)
else if level(e)+1==level(e^f) **then**
  u_f^- = transform-elementdof-to-facedof(u^e,f)
  for all s $\in$ subfaces(f) **do**
    u_s^- = interpolate(u_f^-,s)
    u_s^+ = transform-elementdof-to-facedof(u^{e^f},f)
    flux_s = numerical-flux(u_s^-, u_s^+)
  end for
  flux = integral-projection(flux_s)
else if level(e)==level(e^f)+1 **then**
  s = subface-index (f)
  u^- = transform-elementdof-to-dof(u^e,f)
  u^+ = transform-elementdof-to-dof(u^{e^f},f)
  u_s^+ = interpolate(u_f^+,s)
  flux = numerical-flux(u^-, u_s^+)
end if

5.5 Numerical tests

We validate the convergence of the adaptive scheme. We use a smoothed indicator function as initial conditions. For this purpose, we define the functions

$$h(x) = \begin{cases} e^{-1/x} & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{88}$$

$$g(x) = \frac{h(1-x)}{h(x) + h(1-x)} \tag{89}$$

The function g is smooth as it is the ratio of smooth functions with the denominator greater than 0. Furthermore $g(0) = 1$ and $g(1) = 0$. For a center $\vec{x}_c$, inner radius r_i and outer radius r_o, we define the initial function u_0 by:

$$u_0(\vec{x}) = I_{smooth}(|\vec{x} - \vec{x}_c|) \tag{90}$$

with the one dimensional smoothing function

$$I_{smooth}(x) = \begin{cases} 1, & x \leq r_i \\ g(\frac{x-r_i}{r_o-r_i}), & r_i < x < r_o \\ 0, & x \geq r_o \end{cases} \tag{91}$$

We simulate the advection of the initial function with a constant velocity field in x-direction up to T=1 with CFL=0.05 on the line $[0, 1]$ and on the square $[0, 1]^2$. We chose $r_i = 0.2$ and $r_o = 0.3$. For the center, we use $\vec{x}_c = (0.5)$ in 1D and $\vec{x}_c = (0.5, 0.5)$ in 2D. We display the one-dimensional initial function in Figure 23.

We compare the simulation on a mesh of uniform refinement level l with the simulation on an adaptive mesh with maximum refinement level l and minimum refinement level $l - 2$. We compare all of the refinement criteria described above.

We display the L^2-error for linear polynomials for both the 1D and the 2D case in Figure 24. First of all, we realize that even the uniform refinement only achieves its convergence order starting at level 8. This is because of the sharp increase of the smoothing function, which requires a high refinement level to be resolved.

Both the mass criterion and the relative min-max criterion have the problem, that they do not resolve the region of high curvature of the initial function. This is why they don't achieve much of an improvement against a uniform simulation of level $l - 2$. The artificial criterion on the other hand is chosen so that it reaches the maximum refinement level in all regions, where the solution function does not equal 0 or 1. Thus, it also resolves the regions of high curvature and achieves virtually the same error as the uniform refinement.

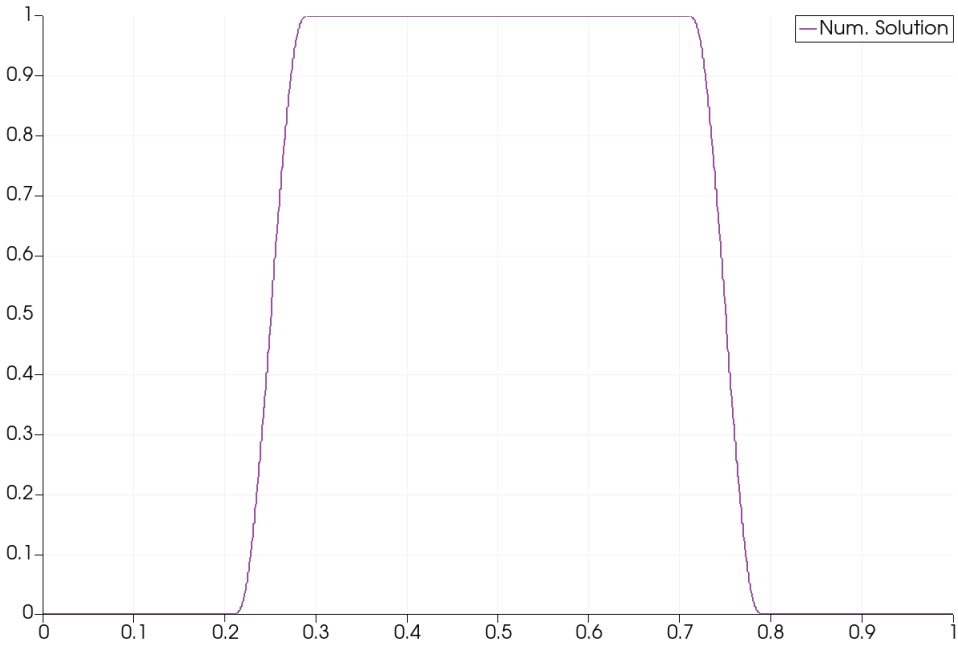

Figure 23: The smoothed indicator function in 1D that we consider for the adapt tests.

For the two-dimensional case, we do not reach a high enough refinement level, to completely resolve the interesting features of the initial function. Thus, we get almost the same error for all simulations.

It is important to realize though, that we start to see improvements in the runtime of our application. We display the needed time for the one-dimensional run and the two-dimensional run in Figure 25. In the one-dimensional case, the number of elements is not drastically decreased for the adaptive meshes, as we only save a factor of 2 for each coarsened element. Therefore, the overhead for our adaptive algorithms overshadows the decrease in elements. The adaptive simulations only become faster around level 9.

This is different for the two-dimensional case. Here, four elements are coarsened together and the region of interest becomes smaller in relation to the size of the physical domain. Thus, we already achieve the same running times for the adaptive and the uniform algorithms on the maximum refinement level 4. On level 7, the uniform simulation needs a factor of 2.87 longer the simulation with the artificial refinement criterion. We expect this improvement to be even higher in three dimensions and will investigate this further in the future.

We display the results for cubic polynomials in Figure 26 and see similar results as for the linear case. The artificial refinement criterion and the

uniform simulation achieve optimal convergence. The mass criterion and the relative min-max criterion do not resolve the regions of high curvature and essentially achieve the error of the of the uniform simulation of level $l-2$ in 1D.

The decrease in runtime displayed in Figure 27 is even higher than for the linear polynomials. At maximum refinement level 7, the uniform simulation took 3.87 times longer than the adaptive simulation with the artificial refinement criterion.

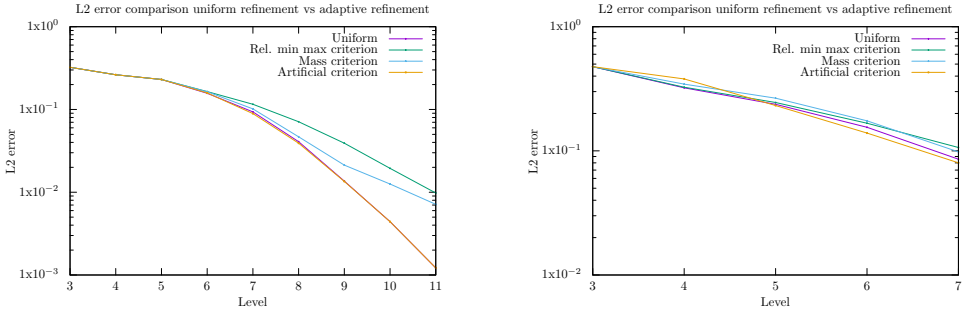

Figure 24: L^2-error of the simulation with linear polynomials on a uniform mesh of level l and the errors for the adaptive simulation with maximum refinement level l and minimum refinement level $l-2$ for the different refinement criteria.

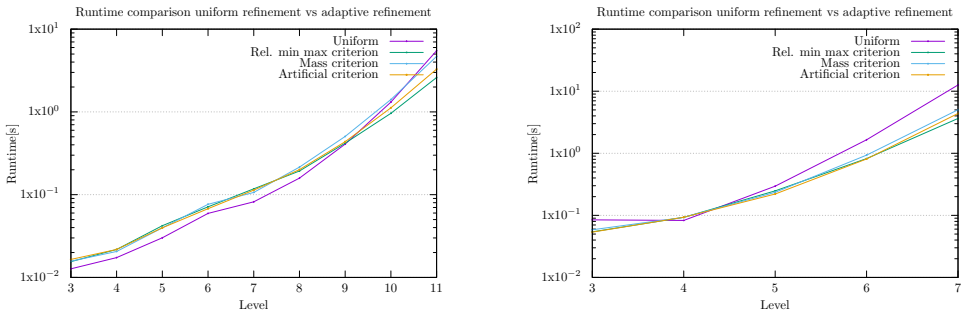

Figure 25: Runtime of the simulation with linear polynomials on a uniform mesh of level l and the runtimes for the adaptive simulation with maximum refinement level l and minimum refinement level $l-2$ for the different refinement criteria.

After validating the adaptive scheme, we use it to simulate the insertion of a tracer into a tube of the form of a cylinder ring. We use the geometry that we introduced in subsection 3.11 together with the spiraling velocity field. We start the simulation with no concentration at all and use the smoothed indicator described above function as a source function. We display images at different timesteps in Figure 28

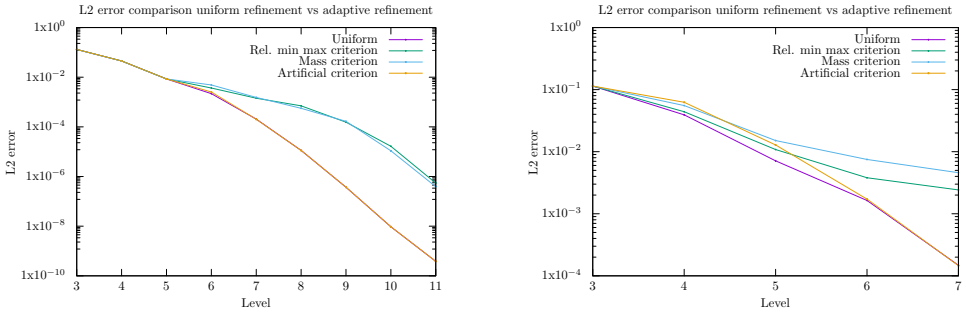

Figure 26: L^2-error of the simulation with cubic polynomials on a uniform mesh of level l and the errors for the adaptive simulation with maximum refinement level l and minimum refinement level $l-2$ for the different refinement criteria.

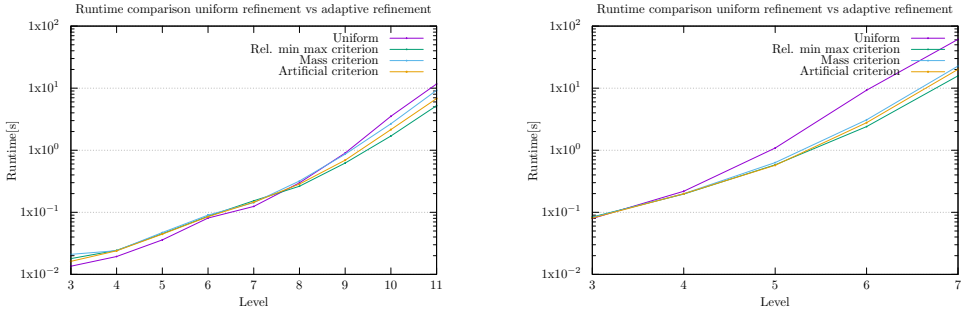

Figure 27: Runtime of the simulation with linear polynomials on a uniform mesh of level l and the runtimes for the adaptive simulation with maximum refinement level l and minimum refinement level $l-2$ for the different refinement criteria.

Summary

- We have introduced and implemented a mortar-based method to extend the discontinuous galerkin method to adaptive meshes.
- There is no loss of accuracy for an artificial refinement criterion, but the investigation of a-posteriori and other refinement criteria for these methods is an important part of further research.
- There was a significant decrease of the runtime in 2D, which we expect to be even greater for simulations in 3D.

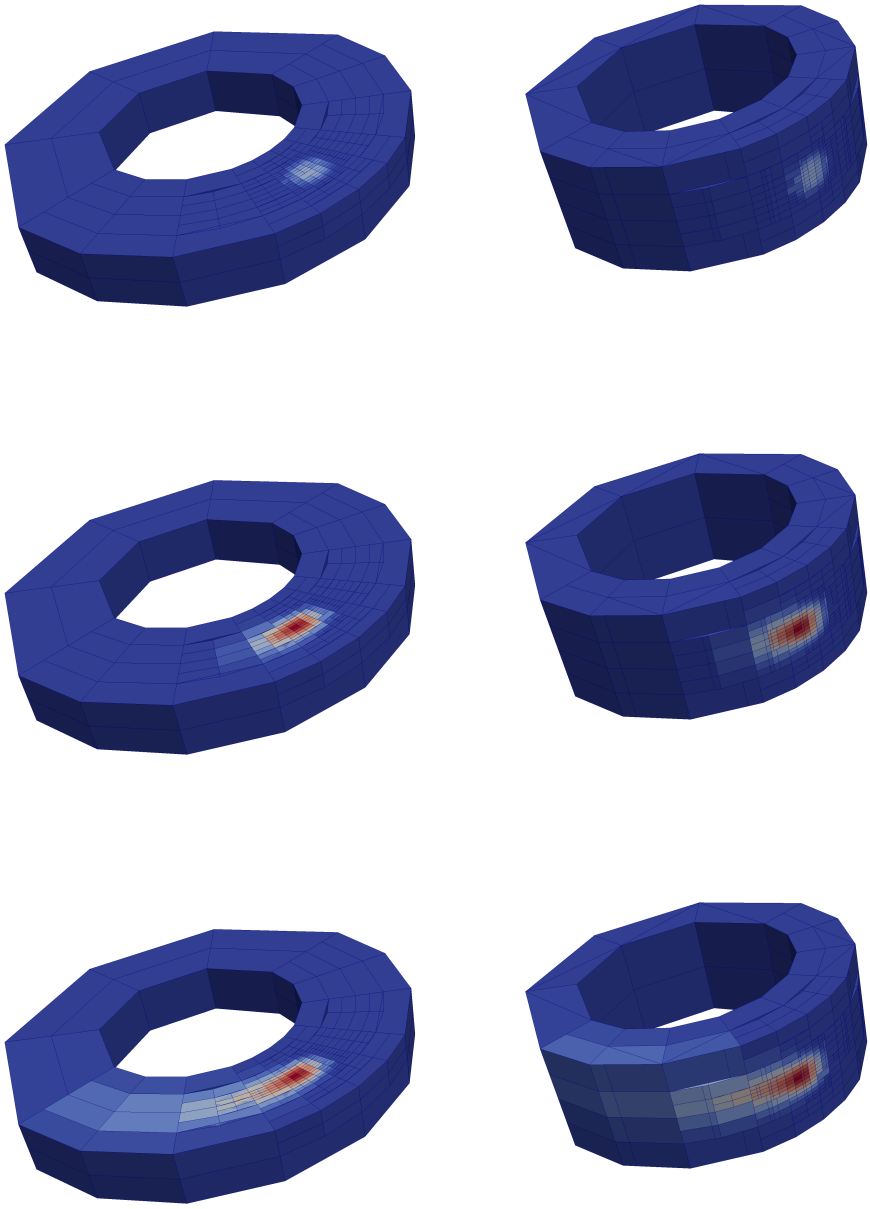

Figure 28: Simulation of the advection and diffusion of a tracer injected into a cylinder ring with spiraling velocity field on an adaptive mesh. Left: Cylinder ring cut in half at $z = 0.5$. Right: a cylindrical cut at $r = 1.5$. Top: $t = 0.01$, Middle: $t = 0.06$, Bottom: $t = 0.25$.

6 Parallelization

In the following chapter, we discuss the parallelization of the DG and LDG method. First, we portray the usage of the parallel grid and data management library `t8code`. Next, we discuss the modifications needed to parallelize the methods on uniform grids. Afterwards we explore some additional peculiarities that we need to consider when we parallelize our methods on adaptive meshes. Finally, we investigate the scalability of our parallel implementation on the JUWELS [44] supercomputer in Jülich.

6.1 Parallelization with `t8code`

In section 2 we introduced `t8code` as a grid management library for massively parallel adaptive meshes. `t8code` uses parallelization based on the MPI standard. Thus, each MPI rank possesses its own distributed memory.

We can share data between different MPI ranks by explicitly communicating linear arrays of data. Collecting data from every MPI rank and distributing it to all MPI ranks is one of the MPI operations needing the most time. We want to avoid these so-called All-to-All communications. If possible, we want to directly communicate information between two involved MPI ranks in a so-called One-to-One communication.

To achieve this, `t8code` uses a ghost layer in which all face-connected elements to a local element are constructed and their corresponding MPI rank is saved. After constructing the ghost layer, `t8code` provides an interface to exchange data for these ghost elements.

For this interface, the data has to be arranged in an array in the order of the SFC index for all local elements. When we construct a forest in `t8code` with a ghost layer, it continues this enumeration with all of the ghost elements. Thus, we provide `t8code` with an array of total size $E_{local}+E_{ghost}$, where E_{local} is the number of local elements and E_{ghost} is the number of ghost elements. The first E_{local} entries of the array correspond to the local data. After calling the ghost exchange interface of `t8code`, the last E_{ghost} entries of the array are filled with the data of the ghost elements.

Additionally, when we query `t8code` for the face neighbors of an element, we get their data indices $idata$, with $0 < idata < E_{local} + E_{ghost}$. We can use this index to access the degrees of freedom(DOFs) and geometry data for local and ghost elements.

6.2 Parallelization on uniform meshes

The computation of the element mass and stiffness matrix only need information their own degrees of freedom and geometry information. Hence, no communication has to take place. Only the calculation of the numerical

fluxes requires communication. We use the interface of `t8code` to exchange the element DOFs and face geometry data of neighboring elements.

After this ghost exchange, a mortar only needs to know the data index *idata* to access the DOFs of the neighbor element and does not need to take into account whether it is a local or a ghost element.

Currently, we only start the communication of the DOFs of ghost elements when we want to calculate the numerical fluxes. The MPI standard supports the splitting of the start of communication and the end of communication and allows for computation to take place in the meantime. `t8code` also supports this ability, but we have not yet adopted this approach for our solver.

Since we could perform the application of the element mass and stiffness matrices during the time in which the ghost dofs are communicated, we expect an additional increase in performance with this addition to our solver.

6.3 Parallelization on adaptive meshes

We use the load balancing capabilities of `t8code` to ensure similar workload on each CPU core. Hence, after each time that the mesh is adapted, we also call the high level method partition in `t8code`. This ensures that the mesh is distributed among the processes with approximately the same number of elements on each process.

Additionally, we can use an interface to partition the DOFs and precalculated geometry data onto the correct processes. Each process needs to call this function with an array that is filled with the data of its local elements before partition and an array with a size of the number of local elements after the partition. After completion of the partition_data algorithm, the second array is filled with the data of the new local elements in the order of the SFC-index.

In the calculation of the numerical fluxes on a mortar, we anchored the mortar to the bigger of the adjacent elements to avoid double computation of the numerical fluxes. In the parallel computation, only some of the subfaces may belong to the local MPI rank. In this case, the local mortar for this face only needs to compute the numerical fluxes on the local subfaces. Only MPI rank that contains the bigger element needs to compute the numerical fluxes on all subfaces and uses integral projection to compute the numerical flux on the bigger face.

Ideally, executing a simulation sequentially or in parallel leads to the same exact results. This is currently not the case for `t8code`. The adapt step can cause different behaviour in parallel than in sequential mode, since `t8code` has not yet implemented partition for coarsening.

In particular, a family of elements is only considered for coarsening if all of its elements are located on the same process. Therefore, a family of elements that is partitioned onto more than one process in parallel mode is

never coarsened, even if it is coarsened in sequential mode.

In practice this has lead to a minuscule increase in elements during the simulation. Furthermore, even if the mesh at the end was the same for a sequential and a parallel simulation, differences in the coarsening of the mesh during the simulation lead to negligible differences in the solution.

6.4 Numerical tests

We perform runtime studies on the JUWELS supercomputer in Jülich. JUWELS is a Bull Sequana X1000 system consisting of 2,271 compute nodes, each node with 96 GB RAM and two 24-core Intel Xeon SC 8168 CPUs running at 2,7GHz. We use one MPI rank per core throughout.

Strong and weak scaling on uniform meshes

We investigate the scaling properties of our algorithm on uniform grids. To this end we examine both its strong and weak scaling.

For the strong scaling, we simulate the advection of the cosine product that we used for the convergence tests in subsection 3.11 with polynomials of degree 3 and Runge-Kutta timestepping methods of order 4. We fix a level and repeatedly double the number of processes. Thus, we expect the runtime to be cut in half. For the weak scaling, we increase the level by one. We run all simulations with a fixed CFL number. Since Δx decreases by a factor of 2, the timestep Δt increases by a factor of 2. The number of elements increases by a factor of 2^d. Thus with perfect strong scaling, we expect that using 2^{d+1} times the number of processes achieves the same runtime on a simulation where the refinement level is increased by one.

We display the results in Figure 29. As long as the number of elements per process is big enough, we achieve almost perfect strong scaling. When the number of elements per MPI rank decreases below 100, the overhead gets too big. We also achieve almost perfect weak scaling, as increasing the level by one and the number of processes by 2^{d+1} only slightly increases the runtime.

Strong scaling on adaptive meshes

To finish the investigation of the scaling properties of our algorithm, we test the strong scaling on a curved adaptive grid in 3 dimensions on 384 to 12288 CPU cores. We simulate the advection and diffusion of a smoothed indicator function as described in subsection 5.5. As a grid, we use the cylindrical mesh described in subsection 3.11, combined with the spiraling velocity field and a diffusion coefficient $a = 10^-5$. For the smoothed indicator function, we use an inner radius of $r_i = 0.1$ and an outer radius of $r_o = 0.2$, with the center in $(1.5, 0, 0.5)$. We use the mass refinement criterion.

Since the scaling of our algorithms already becomes evident for a short time simulation and because of a restriction of resources, we don't simulate a whole rotation through the cylinder. We simulate up to $T = 0.005$, which results in about 100 timesteps with the the chosen CFL number. Additionally, we adapt and partition the mesh 4 times during the simulation. The simulation starts with about 750,000 elements, in the end we have about 1,400,000 elements.

In Figure 30, we show the runtime plotted against the number of processes. We observe almost perfect scaling over the whole range, only losing about a factor of about 1.5 for the simulation without the initialization time. In the initialization phase, the grid is repeatedly adapted and partitioned to achieve a grid that resolves the initial function. Because the mesh changes a lot during this phase, a lot of communication takes place between processes. For simulations over a longer time range, the initialization time matters less.

In Figure 31 we split the runtime in its parts. It is evident that a big factor restricting the scalability of the simulation is the exchange of DOFs for ghost elements.

As we discussed before, we believe that there is still a lot of improvement possible in using the time during which the communication of the DOFs takes place. Currently the calculation of the mass and stiffness matrix, which don't need ghost values, take place independently of the calculation of the numerical fluxes and boundary integrals. Thus the communication of ghost values only starts when the calculation of the mass and stiffness matrix is finished. We wait for the end of the communication before we start calculating the numerical fluxes. We will improve this process sending the ghost values before we calculate the application of the mass and stiffness matrix.

Additionally, we could begin computing the numerical fluxes for faces where all neighboring elements are processor local. Afterwards, we expect the communication of the ghost values to be finished. `t8code` supports the MPI capabilities of splitting the start and end of communication with the option to use the meantime for calculations. We plan on integrating these capabilities in the `t8dg` software in the future.

Another interesting aspect is the increase in runtime of the ghost and partition_data algorithms for the runs on 3072, 6144 and 12288 CPU cores. We believe that this is related to the way we executed the runs on JUWELS. We did 2 runs, one on 32 nodes and one on 256 nodes. We used a quarter, a half and the full number of CPU cores on each Node to achieve the different number of MPI tasks. As a result, the communication in the bigger run has to take place between a bigger number of nodes in total.

The 2 different runs also explain the sharp increase in the initialization time for 3072 cores. We did not run a program to warm up the CPU cores before starting our scaling runs. Thus, the initialization phase of the first simulation in a run falls into that process of warming up the CPU cores.

For future runs, we will eliminate this problem to obtain more consistent timing data.

Finally, we observe a sharp increase in the balance algorithm for 12288 CPU cores. We suspect that this is the first time that the ripple effect of balance crosses multiple processor boundaries. Balance is an iterative algorithm that is reported by the authors to be the most expensive AMR algorithm in `t8code` [39].

We did not yet investigate the weak scaling of the adaptive methods in this thesis. We plan to do so in the future. A challenge in investigating the weak scaling of adaptive methods is that the number of elements does not necessarily increase by a factor of 2, when we increase the maximum refinement level by 1. This challenge is overcome by considering the average time per element.

Summary

- Until partition for coarsening is implemented, parallel and sequential simulations with the same parameters can lead to slightly different results.
- We obtain almost perfect strong and weak scaling properties for uniform meshes with more than 100 elements per process.
- We can use the time needed for communication for local computations to further increase the efficiency of our algorithms.

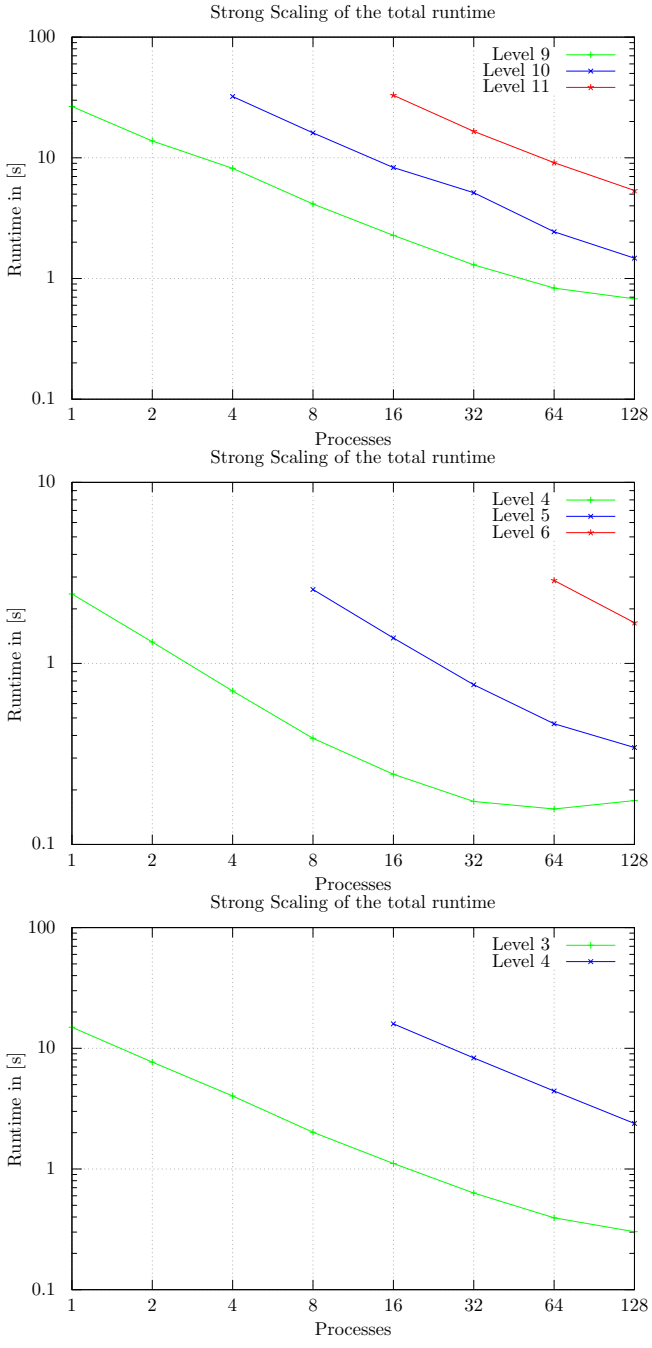

Figure 29: Strong scaling in 1 up to 3 Dimensions. We also achieve good weak scaling properties, as a fixed CFL constant implies an increase in the problem complexity by a factor of 2^{d+1} when increasing the level by 1.

Figure 30: Strong scaling of the advection-diffusion equation on a cylinder ring mesh on up to 12288 CPU cores

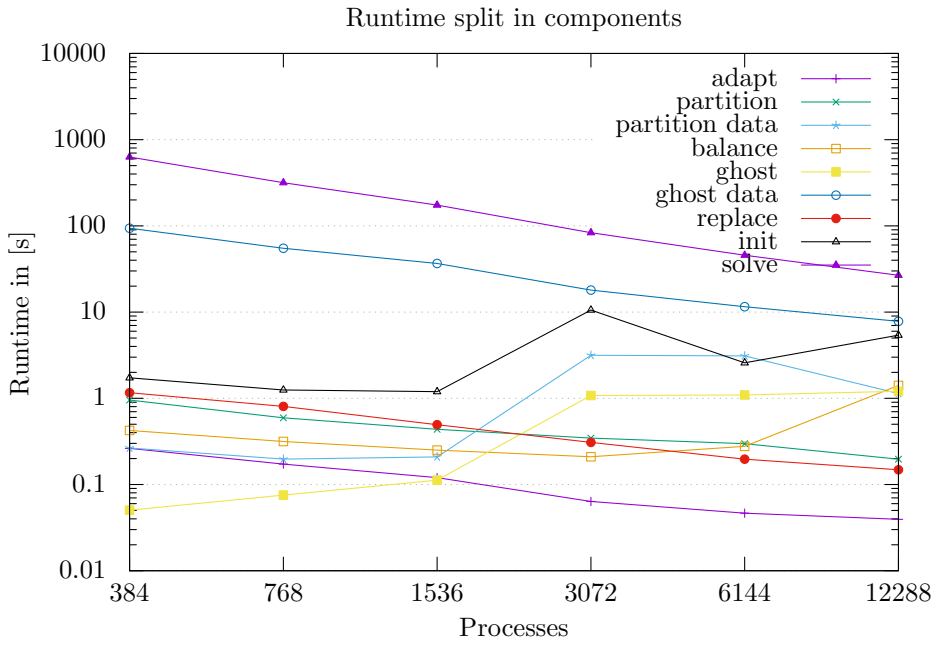

Figure 31: Runtime of the scaling test split into its components

7 Conclusion and Outlook

Conclusion

In this thesis, we developed a new modular approach for the Discontinuous Galerkin (DG) method and implemented it in the application `t8dg`. We utilize this approach to simulate the linear advection of a tracer in curved geometries. Additionally, we took advantage of the modular structure of our application to extend it to the Local Discontinuous Galerkin (LDG) method for the advection diffusion equation.

Furthermore, we found two ways to efficiently reduce the computational cost of these methods. First, we combine the DG and LDG method with adaptive meshes, which significantly reduces the number of elements. Thus, the runtime and the memory footprint are significantly reduced as well. Second, we used the grid and data management library `t8code` to achieve a highly effective parallel implementation of our algorithms. We showed that our application scaled perfectly if the problem remains large enough for each process.

Both methods can also be combined, leading to a fast and efficient way to simulate the advection and diffusion of a tracer in a given velocity field. In the following, we summarize the most important results of this thesis.

To establish that the DG method on one process is implemented correctly, we ran a substantial amount of time and space convergence tests with different orders in one dimension. The results of these tests showed that for polynomials of degree at most k and explicit low memory Runge-Kutta schemes of order $k+1$, the expected $(k+1)st$ order of convergence in the L^2 norm is achieved.

With this knowledge, we extended our tests into two and three space dimensions. We combined tensor products of polynomials of degree k with the Runge-Kutta schemes of order $k+1$ and achieved optimal $(k+1)st$ order of convergence. Finally, we tested our implementation on meshes with curved boundaries and given analytic geometries and achieved the same order of convergence.

Afterwards, we ran tests for the advection-diffusion equation with dominating advection and for the purely parabolic diffusion equation. We compared the convergence properties of the central and the alternating flux, two commonly used numerical fluxes for the diffusive part.

In the case of dominating advection, the choice of flux only marginally affected the error, and optimal order of convergence was achieved for both numerical fluxes.

For the purely diffusive simulation, the choice of flux did matter. While the alternating flux achieved optimal order of convergence for all polynomial degrees, the central flux only lead to third order convergence for polynomials of degree 3. This is in line with the theoretical results proving $k+1$st order

convergence for degree k polynomials with k even, and only kth order of convergence for k odd. Linear polynomials achieved second order convergence, improving on the theoretical results.

Next, we proceeded by implementing the adaptive version of the DG and LDG method presented in this thesis. The subsequent numerical investigation showed the importance of the refinement criterion. The relative min-max criterion and the mass criterion were not able to completely resolve the interesting features of the initial function. Thus, we tested the scheme with an artificial refinement scheme that knew about the interesting features in advance. With this refinement criterion we achieved almost the same error as in the uniform simulation. Meanwhile, the resulting reduction in the number of elements also lead to a significant reduction in the runtime.

Therefore, with an appropriate refinement criterion, our adaptive method is very successful in reducing computational cost.

To further decrease computation time, we combined both the uniform and the adaptive solver with a parallel computation of the numerical fluxes. As the grid parallelization is handled by `t8code`, we used its data management capabilities to distribute degrees of freedom and geometry information of ghost elements.

We tested the scaling of our uniform algorithms on up to 128 CPU cores and achieved almost perfect strong and weak scaling for problems that were large enough on each process. Afterwards, we ran a strong scaling test on up to 12288 CPU cores for our adaptive algorithm and achieved near perfect scaling. The scaling over a range of 384 to 12288 CPU cores makes us confident for planned scaling tests exhausting the full size of the JUWELS supercomputers.

In conclusion, we have developed a highly scalable, modular DG solver on adaptive meshes. We demonstrated its convergence, scalability and handling of adaptive meshes.

Outlook:

The developement of parallel adaptive DG and LDG methods in the `t8dg` application does not stop with this thesis. There are several natural extensions to the application. We will list some of them below.

We have laid the foundation for the treatment of hybrid meshes in this thesis. With our modular approach, new element types can be supported by implementing a respective functionbasis and quadrature. As a next step, we plan on implementing these for triangular elements.

Additionally, we are interested in integrating prismatic elements into our application, since these are traditionally used in numerical weather prediction for meshes covering the atmosphere above earths surface. We already implemented the tensor product construction for the functionbasis and quadrature of quadrilateral and hexahedral elements. Therefore, we can use this

construction for the tensor product of a triangle with a line to obtain the needed modules for prismatic elements.

The ultimate goal is to support all element types implemented in `t8code`, so that our solver is fully flexible to handle hybrid meshes in 2D and 3D.

A different direction, but an equally important part of our further research, is the support of non-linear partial differential equations. The biggest obstacle is the integration of a suited generalized slope-limiter on multidimensional adaptive grids.

Another aspect that we will explore is the development of an appropriate refinement criterion. We have seen in this thesis that the refinement criterion plays an important role in the construction of effective adaptive schemes. For the extension of our scheme to nonlinear PDEs, the capturing of shocks with elements of a high resolution will be an important challenge.

The mortar-based approach to calculating numerical fluxes allows for hp-adaptive methods. With these methods, higher order polynomials are used in regions where the solution is smooth. In contrast, higher refinement levels are used in regions where the solution displays shocks.

Furthermore, the modular structure of `t8dg` allows us to investigate different kinds of numerical fluxes and their influence on the accuracy and stability of the DG and LDG methods.

Finally, we have seen that the explicit time-stepping schemes are restricted by a CFL-type criterion. This becomes especially restrictive for simulations where the diffusion dominates, on meshes with elements of different sizes, and for simulations in which the scale of the involved processes differs between directions. An example of the last aspect is the simulation of earths atmosphere, which is often meshed with a higher resolution in the vertical direction because of faster gravitational vertical processes. An interesting approach to this problem is to solve the vertical direction implicitly in time while using explicit time integration in the horizontal direction.

List of Figures

List of Algorithms

References

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55 of *National Bureau of Standards Applied Mathematics Series*. U.S. Government Printing Office, Washington, D.C., 1964.

[2] John David Anderson and J. Wendt. *Computational fluid dynamics*, volume 206. Springer, 1995.

[3] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[4] Abdelkader Baggag, Harold Atkins, and David Keyes. *Parallel implementation of the discontinuous Galerkin method*. Institute for Computer Applications in Science and Engineering, NASA Langley, 1999.

[5] Guido Baruzzi, Wagdi Habashi, J. Guevremont, and M. Hafez. A second-order finite element for the solution of the transonic euler and navier-stokes equations. *International Journal for Numerical Methods in Fluids*, 20:671 – 693, 1995.

[6] F. Bassi and S. Rebay. A High-Order Accurate Discontinuous Finite Element Method for the Numerical Solution of the Compressible Navier-Stokes Equations. *Journal of Computational Physics*, 131(2):267–279, 1997.

[7] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000.

[8] Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.

[9] Charles E. Baukal Jr, Vladimir Gershtein, and Xianming Jimmy Li. *Computational fluid dynamics in industrial combustion*. CRC press, 2000.

[10] Faker Ben Belgacem. The mortar finite element method with lagrange multipliers. *Numerische Mathematik*, 84(2):173–197, 1999.

[11] Marsha Berger and Randall Leveque. An adaptive cartesian mesh algorithm for the euler equations in arbitrary geometries. In *9th Computational Fluid Dynamics Conference*, page 1930, 1989.

[12] Marsha J. Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[13] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[14] Jiri Blazek. *Computational fluid dynamics: principles and applications*. Butterworth-Heinemann, 2015.

[15] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[16] Carsten Burstedde and Johannes Holke. Coarse mesh partitioning for tree based AMR. In review, 2016.

[17] Carsten Burstedde and Johannes Holke. A tetrahedral space-filling curve for nonconforming adaptive meshes. *SIAM Journal on Scientific Computing*, 38(5):C471–C503, 2016.

[18] Carsten Burstedde, Johannes Holke, and Tobin Isaac. On the number of face-connected components of Morton-type space-filling curves. *Foundations of Computational Mathematics*, pages 1–26, 2018.

[19] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. P4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[20] J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley-Interscience, USA, 1987.

[21] John Charles Butcher. A history of runge-kutta methods. *Applied numerical mathematics*, 20(3):247–260, 1996.

[22] D. Calhoun and C. Burstedde. Highly scalable adaptive mesh refinement for natural hazards modeling. In *AGU Fall Meeting Abstracts*, volume 2018, pages DI23A–06, 2018.

[23] Paul Castillo, Bernardo Cockburn, Ilaria Perugia, and Dominik Schötzau. An a priori error analysis of the local discontinuous galerkin method for elliptic problems. *SIAM Journal on Numerical Analysis*, 38(5):1676–1706, 2000.

[24] Bernardo Cockburn, Suchung Hou, and Chi-Wang Shu. The Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws. IV: The Multidimensional Case. *Mathematics of Computation*, 54(190):545, 1990.

[25] Bernardo Cockburn, George E Karniadakis, and Chi-Wang Shu. *Discontinuous Galerkin methods: theory, computation and applications*, volume 11. Springer Science & Business Media, 2012.

[26] Bernardo Cockburn, San-Yih Lin, and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-dimensional systems. *Journal of Computational Physics*, 84(1):90–113, 1989.

[27] Bernardo Cockburn and Chi-Wang Shu. The Runge-Kutta local projection P1-discontinuous-Galerkin finite element method for scalar conservation laws. *IMA Preprint Series #388, University of Minnesota*, 1988.

[28] Bernardo Cockburn and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. *Mathematics of Computation*, 52(186):411–411, 1989.

[29] Bernardo Cockburn and Chi-Wang Shu. The Local Discontinuous Galerkin Method for Time-Dependent Convection-Diffusion Systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[30] Bernardo Cockburn and Chi-wang Shu. The RungeKutta Discontinuous Galerkin Method for Conservation Laws V. *Journal of Computational Physics*, 141(2):199–224, 1998.

[31] Phillip Colella, John Bell, Noel Keen, Terry Ligocki, Michael Lijewski, and Brian Van Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. In *Journal of Physics: Conference Series*, volume 78, page 012013. IOP Publishing, 2007.

[32] Lukas Dreyer. `t8dg`: DG solver on parallel adaptive meshes, 2021. `http://github.com/lukasdreyer/t8dg`, last accessed February 20th, 2021. Currently private, made public in the near future.

[33] Herbert Edelsbrunner et al. *Geometry and topology for mesh generation*. Cambridge University Press, 2001.

[34] Vamsi Ganti, Mark M. Meerschaert, Efi Foufoula-Georgiou, Enrica Viparelli, and Gary Parker. Normal and anomalous diffusion of gravel tracer particles in rivers. *Journal of Geophysical Research: Earth Surface*, 115(F2), 2010.

[35] AD Gosman. Developments in industrial computational fluid dynamics. *Chemical Engineering Research and Design*, 76(2):153–161, 1998.

[36] Yi Heng, Lars Hoffmann, Sabine Griessbach, Thomas Rössler, and Olaf Stein. Inverse transport modeling of volcanic sulfur dioxide emissions using large-scale simulations. *Geoscientific model development*, 9(4):1627–1645, 2016.

[37] L. Hoffmann, T. Rössler, S. Griessbach, Yi Heng, and O. Stein. Lagrangian transport simulations of volcanic sulfur dioxide emissions: impact of meteorological data products. *Journal of geophysical research / Atmospheres*, 121(9):4651–4673, 2016.

[38] L. Hoffmann, X. Xue, and M. J. Alexander. A global view of stratospheric gravity wave hotspots located with atmospheric infrared sounder observations. *Journal of Geophysical Research: Atmospheres*, 118(2):416–434, 2013.

[39] Johannes Holke. *Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2018. https://bonndoc.ulb.uni-bonn.de/xmlui/handle/20.500.11811/7661.

[40] Johannes Holke and Carsten Burstedde. `t8code`: Parallel AMR on hybrid non-conforming meshes, 2015. `http://github.com/holke/t8code`, last accessed February 14th, 2021.

[41] Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-cost parallel algorithms for 2:1 octree balance. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*, pages 426–437, 2012.

[42] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[43] Claes Johnson and Juhani Pitkäranta. An analysis of the discontinuous galerkin method for a scalar hyperbolic equation. *Mathematics of computation*, 46(173):1–26, 1986.

[44] Jülich Supercomputing Centre. JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre. *Journal of large-scale research facilities*, 5(A135), 2019.

[45] David Knapp. Adaptive Verfeinerung von Prismen. Bachelor's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[46] David Knapp. A space-filling curve for pyramidal adaptive mesh refinement. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2020.

[47] Donald L Koch and John F Brady. A non-local description of advection-diffusion with application to dispersion in porous media. *Journal of Fluid Mechanics*, 180:387–403, 1987.

[48] David A. Kopriva. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[49] W. Kutta. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeit. Math. Phys.*, 46:435–53, 1901.

[50] P Lesaint and Pierre-Arnaud Raviart. On a finite element method for solving the neutron transport equation. *Mathematical aspects of finite elements in partial differential equations*, (33):89–123, 1974.

[51] Simone Marras, James F. Kelly, Margarida Moragues, Andreas Müller, Michal A. Kopera, Mariano Vázquez, Francis X. Giraldo, Guillaume Houzeaux, and Oriol Jorba. A Review of Element-Based Galerkin Methods for Numerical Weather Prediction: Finite Elements, Spectral Elements, and Discontinuous Galerkin. *Archives of Computational Methods in Engineering*, 2016.

[52] Steven J Owen. A survey of unstructured mesh generation technology. *IMR*, 239:267, 1998.

[53] Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890.

[54] Todd E Peterson. A note on the convergence of the discontinuous galerkin method for a scalar hyperbolic equation. *SIAM Journal on Numerical Analysis*, 28(1):133–140, 1991.

[55] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical Mathematics*. II. Springer Verlag, New York, 2000.

[56] Wm H Reed and TR Hill. Triangularmesh methodsfor the neutron-transportequation. *Los Alamos Report LA-UR-73-479*, 1973.

[57] William E Schiesser. *The numerical method of lines: integration of partial differential equations*. Elsevier, 2012.

[58] Gordon Scott and Philip Richardson. The application of computational fluid dynamics in the food industry. *Trends in Food Science & Technology*, 8(4):119–124, 1997.

[59] Bruno Seny, Jonathan Lambrechts, Thomas Toulorge, Vincent Legat, and Jean-François Remacle. An efficient parallel implementation of explicit multirate runge–kutta schemes for discontinuous galerkin computations. *Journal of Computational Physics*, 256:135–160, 2014.

[60] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry*, 22(1-3):21–74, 2002.

[61] Mitsuharu Terashima, Rajeev Goel, Kazuya Komatsu, Hidenari Yasui, Hiroshi Takahashi, YY Li, and Tatsuya Noike. Cfd simulation of mixing in anaerobic digesters. *Bioresource technology*, 100(7):2228–2233, 2009.

[62] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson education, 2007.

[63] Qiang Zhang and Chi-Wang Shu. Error estimates to smooth solutions of runge–kutta discontinuous galerkin methods for scalar conservation laws. *SIAM Journal on Numerical Analysis*, 42(2):641–666, 2004.

[64] Qiang Zhang and Chi-Wang Shu. Error estimates to smooth solutions of runge–kutta discontinuous galerkin method for symmetrizable systems of conservation laws. *SIAM Journal on Numerical Analysis*, 44(4):1703–1720, 2006.