FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Diplomová práce

Discontinuous Galerkin method, its analysis and implementation into SfePy package

Plzeň 2020 Tomáš Zítka

Prohlášení

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně s použitím odborné literatury a pramenů, uvedených v seznamu, který tvoří přílohu této práce.

V Plzni dne 20. 7. 2020

Tomáš Zítka

Poděkování

Děkuji svému školiteli Ing. Robertu Cimrmanovi, Ph.D. za trpělivost a cenné rady při vedení této práce.

Abstract

This diploma thesis describes the theory behind the discontinuous Galerkin finite element method (DG FEM) and subsequently the implementation of the method into numerical software package SfePy (Simple finite elements in Python). SfePy uses the term based representation to specify a solved equation. Several new terms needed in DG FEM formulation were implemented along with an internal representation of the discretization. Namely the linear advection flux term, the general hyperbolic flux term, the diffusion flux term, and the diffusion penalty term. To enable solving transient equations two explicit time-stepping solvers, the forward Euler solver and the TVD Runge-Kutta of the 3rd order solver were implemented. Moreover, the moment limiters for 1D and 2D transient problems were also implemented. This implementation was then used in eight examples to test the convergence of the method and illustrate the effects and interactions of the diffusion penalty term and the limiter. The diffusion penalty term proves to be necessary to overcome the discontinuity of the method in problems that mandate a continuous solution. The limiter causes significant artificial diffusion but keeps oscillation occurring in the high order approximations manageable.

Abstrakt

Tato diplomová práce popisuje teorii za nespojitou Galerkinovou metodou konečných prvků (DG MKP) a následně její implementaci do numerického software SfePy (Simple finite elements in Python). SfePy používá k reprezentaci řešených rovnic jednotlivé, předpřipravené integrální termy. Tato práce popisuje odvození a implementaci několika termů, potřebných k formulaci metody. Jmenovitě jde o lineární advekční term, obecný hyperbolický term, difuzní term a difuzní penaltový term. Spolu s nimi byla do SfePy přidána i potřebná vnitřní reprezentace diskretizace. Pro řešení tranzientních problémů jsou součástí implementace i dva explicitní časové řešiče: dopředný Eulerův řešič a TVD Runge-Kutta třetího řádu. Pro použití v tranzientních problémech je určen tzv. momentový limeter implementovaný pro 1D problémy a 2D problémy s uniformní čtyřúhelníkovou sítí. Tato implementace byla následně použita k řešení osmi příkladů, na kterých testujeme konvergenci metody a vliv limiteru a penaltového členu. Penaltový člen se ukazuje jako nezbytný v problémech, které vyžadují spojité řešení. Limiter způsobuje značnou umělou difuzi avšak zabraňuje oscilacím, které se objevují u metod vyššího řádu.

Contents

List of Examples

List of Figures

Chapter 1

Introduction

In this work, we present the implementation of discontinuous Galerkin Finite Elements Method (DG FEM) in software package *SfePy* and results of experimental measurement of convergence of the method. We consider several model problems, the basic one being a linear advection problem for variable p with constant advection velocity $\vec{a}$

$$\frac{\partial p}{\partial t} - \vec{a} \cdot \nabla p = 0,$$

with a Dirichlet boundary conditions

$$p(t, \vec{x}) = p_D(t, \vec{x}) \text{ for } \vec{x} \text{ in } \partial\Omega,$$

which we gradually generalize by adding diffusion D, source term g and nonlinearity $\vec{f}$

$$\frac{\partial p}{\partial t} + \nabla \cdot \vec{f}(p) - D\Delta p = g.$$

The first main goal of this work is to provide an implementation of the discontinuous Galerkin method, which could be used to empirically study the behavior of the method but also in academic and potential real-world applications and in education. The second main goal is to use this implementation to analyze the behavior of the method when applied to chosen model problems based on the equations above, especially concerning the choice of different fluxes and penalty terms (see below).

The work is divided as follows: The introductory chapter summarizes literature and basic concepts of DG FEM and introduces *SfePy*. In the second chapter, we derive the method for a model problem and explore the theory behind it. The third chapter describes in detail the *SfePy* package and the implementation of the method. The fourth chapter presents the setup and results of numerical experiments measuring convergence. In the concluding chapter, we discuss the results and present suggestions for future work.

1.1 Basic concepts and literature

In continuous or classical finite element discretizations of the partial differential equation, the solution is approximated by a combination of basis functions whose supports span across multiple geometrical elements of the mesh discretizing the computational domain. This enforces continuity of the solution and provides a way of transferring information between the elements. In discontinuous Galerkin FE methods, on the other hand, the basis functions used in the approximation of test and state variables have supports limited to the individual geometrical elements, much like piecewise approximation in finite volume (FV) methods. This leads to compact discretization stencils and allows discontinuities in the approximate solution but also requires fluxes at element

interfaces to be introduced in order to transfer information between elements. As we shall see, these properties of the DG FE methods prove to be useful in some applications and burdens in others.

The discontinuous approximation and compact stencils make DG FEM appealing for multi-domain and multi-physics simulations [11]. The possibility to approximate discontinuous solutions proves useful in modeling so-called shock waves in nonlinear conservation laws with small dissipation [19](see Example 7). Inherent discontinuity of solution, however, brings difficulties for diffusion dominated or otherwise naturally continuous problems and forces introduction of so-called penalty terms (more in Section 2.3.2, [3] and [19]). The introduction of fluxes provides great flexibility of DG FEM and allows a straightforward implementation of conservation laws which endows the method with good stability properties when approximating advection dominated problems. Among disadvantages of the use of fluxes belong a complicated theoretical analysis of the methods and a lack of an exact solution to Riemann problems for high order approximations in individual mesh elements. The use of approximate fluxes for solving Riemann problems with rough initial data with large gradients introduces oscillations not present in FV methods mandating the use of so-called limiters (more in Section 2.7, [11, Sec. 3.2.4] and [18]). The FE nature of DG FEM and the use of fluxes allows the DG FEM to be interpreted both as Galerkin projection onto suitable energy spaces as well as high order classical upwind finite volume schemes [13].

Although studied thoroughly, as literature cited above suggests, DG FE methods still pose research challenges and promise new and potentially useful results for numerical modeling. Among challenges are those mentioned above. One of great promises is the so-called super-convergence observed for certain problems [22] which yet awaits to be leveraged in applications.

1.2 *SfePy* – Simple Finite Elements in Python

Simple finite elements in Python (*SfePy*, `http://sfepy.org/`) is a software package providing FE based methods along with a wide range of tools for defining, solving, and post-processing variety of coupled PDEs in 1D, 2D, and 3D. It can be viewed both as a black-box PDE solver and as a Python package which can be used for building custom application [8]. The code of the package is open-source published under New BSD-3 Clause license [17] and is available on Github [6]. Detailed documentation with many examples can be found in [7].

SfePy can use many FE based terms to build the PDEs to be solved. This approach is reflected in Section 2 where the discretization of the equation is divided into discretization of individual terms, these are then implemented individually in Chapter 3. As of time of writing *SfePy* supports classical FEM and isogeometric analysis (IGA) based FEM and provides tools for setting up, solving and post-processing problems in applications like homogenization of porous media, acoustic waves in thin perforated layers, finite element formulation of Schroedinger equation or flow of a two-phase non-Newtonian fluid medium in a general domain [8].

There are several other software packages implementing DG FEM, some of the currently available codes are: hpGEM [20] (`https://hpgem.org/`) which provides implementation of so-called *hp*-methods in C++; FEniCS project [1] (`https://fenicsproject.org/`) which is well established numerical software build on C/C++; PyFR [24] (`http://www.pyfr.org/`) which is Python based framework for solving advection-diffusion type problems that leverages locality of DG FE methods to run computations efficiently on modern streaming architectures, such as Graphical Processing Units (GPUs)[24].

Chapter 2

Discontinous Galerkin Method

In this chapter, we lay the theoretical background for the concepts necessary to describe the method and then derive the discretization of the advection term $\vec{a} \cdot \nabla(p)$, the general nonlinear hyperbolic term $\nabla \cdot \vec{f}(p)$, the diffusion term $\nabla \cdot (D\nabla p)$ and the source term $g(x)$ occurring in various PDEs, namely, from the simplest one: the linear advection equation with a constant and non-constant velocity, the advection with diffusion and sources, the general nonlinear hyperbolic equation. Using the discretized terms we will formulate the method for these equations. Towards the end of the chapter, we introduce limiters necessary for stabilization of the high order versions of the method.

2.1 Terms and equations

The basic equation we will be concerned with is the partial hyperbolic-elliptic equation for the unknown function p, $p : \Omega \to \mathbb{R}$, where $\Omega \subset \mathbb{R}$ is the physical domain with the boundary $\partial\Omega$, in the stationary form

$$\nabla \cdot \vec{f}(p) - \nabla \cdot (D\nabla p) = g, \tag{2.1.1}$$

where f is a sufficiently smooth vector function $f : \mathbb{R} \to \mathbb{R}^n$, with the Dirichlet boundary conditions

$$p(t, \vec{x}) = p_D(t, \vec{x}) \text{ for } \vec{x} \text{ in } \partial\Omega, \tag{2.1.2}$$

or in the transient form

$$\frac{\partial p}{\partial t} + \nabla \cdot \vec{f}(p) - \nabla \cdot (D\nabla p) = g, \tag{2.1.3}$$

with boundary conditions of the same form and the initial condition

$$p(0, \vec{x}) = p_0(\vec{x}). \tag{2.1.4}$$

For this problem we will be concerned with discretization of the generally nonlinear hyperbolic term covered in Section 2.3.1

$$\nabla \cdot \vec{f}(p), \tag{2.1.5}$$

which also covers discretization of the linear advection term

$$\vec{a} \cdot \nabla p; \tag{2.1.6}$$

the diffusion term covered in Section 2.3.2

$$-\nabla \cdot (D\nabla p); \tag{2.1.7}$$

the source term in Section 2.3.3

$$g; \tag{2.1.8}$$

and finally in Section 2.4 we will treat discretization of the temporal derivative term

$$\frac{\partial p}{\partial t}. \tag{2.1.9}$$

2.2 Finite dimensional discontinuous approximation space

In order to discretize the terms and further the equations we first need to establish the approximation space we will use, similarly to continuous FEM. We start by choosing a suitable computational domain Ω_h which approximates the domain Ω. Since *SfePy* supports simplex and tensor product meshes, we will be concerned with space filling tessellations containing line segments for 1D problems, and only triangles or only quadrangles for 2D problems. The subscript h denotes the minimal diameter of the elements, N will denote the number of elements of Ω_h and individual elements will be denoted by T^k for $k = 0, \ldots, N-1$. Creating this suitable tessellation for an arbitrary computational domain is by no means a trivial task, however for the time being we will assume the selected computational domain and the mesh satisfy all conditions required below. First we define the space of piecewise continuous functions on Ω_h as

$$C^1(\Omega_h) = \left\{ v;\ v|_{T^k} \in C^1 \quad \forall T^k \in \Omega_h \right\}. \tag{2.2.1}$$

and the broken Sobolev space on Ω_h as

$$W^{1,2}(\Omega_h) = \left\{ v;\ v|_{T^k} \in W^{1,2} \quad \forall T^k \in \Omega_h \right\}. \tag{2.2.2}$$

In the finite dimensional discretization we will work in finite dimensional subspaces of $W^{1,2}(\Omega_h)$. On each element T^k we express solution $p(t, \vec{x})$ locally as a linear combination of basis functions

$$p(t, \vec{x})|_{T^k} \approx p_h^k(t, \vec{x}) = \sum_{n=0}^{N_{base}-1} P_n^k \psi_n^k(\vec{x}), \tag{2.2.3}$$

i.e., as a function in the local space

$$V_{T^k} = \mathrm{span}\left\{ \psi_n^k(\vec{x}),\ n = 0, 1, \ldots N_{base} - 1 \right\}, \tag{2.2.4}$$

where N_{base} is the number of basis functions we use in the approximation and hence the dimension of the local approximation space. This number is directly tied to approximation order and is dependent on the type of mesh elements. Additionally we also require that

$$\mathrm{supp}\left\{ \psi_n^k(\vec{x}) \right\} = T^k \quad n \in \{0, 1, \ldots N_{base} - 1\}. \tag{2.2.5}$$

This means that basis functions are localized to individual elements and allow us to represent a discontinuous solution, unlike in the classical FEM, where supports of basis functions overlap, spanning multiple elements and thus enforcing continuity of solution.

In 1D, $\psi_n^k(\vec{x})$ is composed of Legendre polynomials shifted and truncated to interval $[0, 1]$; we denote $L^r(x)$ the Legendre polynomial of order r. These Legendre polynomials are orthogonal and hence the set $\{\psi_n^k(x)|\ \psi_n = L^n,\ n = 0, 1, \ldots N_{base} - 1\}$ forms basis of the N_{base} dimensional polynomial space. We denote $M = N_{base} - 1$ the maximal order of used Legendre polynomials.

In 2D the basis functions $\psi_n^k(\vec{x})$ are composed from Legendre polynomials in such a way that the set $\{\psi_n^k(\vec{x})|\ n = 0, 1, \ldots N_{base} - 1\}$ is orthogonal with respect to the local scalar product

$$(p, v)_{T^k} = \int_{T^k} p \cdot v. \tag{2.2.6}$$

hence forming basis of the N_{base}-dimensional space. The particular shape of $\psi_n^k(\vec{x})$ depends on topology of the mesh elements. For tensor product meshes, i.e., quadrilateral elements we use the straight-forward tensor product of Legendre polynomials. If we denote M the order of approximation, d the dimension of the geometric space (2 in our case), we get quadrilateral element basis functions in the form

$$\psi_n^k(\vec{x}) = L^r(x) L^s(y) \quad r, s = 0, 1, \ldots, M, \tag{2.2.7}$$

and the dimension of the approximation space is

$$N_{base} = (M + 1)^d. \tag{2.2.8}$$

In the case of simplex meshes, the shape of $\psi_n^k(\vec{x})$ is the result of Gram-Schmidt orthogonalization process on the canonical basis

$$\{x^r y^s, \quad r,s = 0,1,\ldots,M \text{ s.t. } r+s \leq M\}, \tag{2.2.9}$$

with respect to scalar product (2.2.6) and its shape is much more elaborate. The Jacobi polynomials are needed to represent the basis. If we denote $J_m^{\alpha,\beta}$ the m-th order Jacobi polynomial, the individual basis functions can be written in the form [15]

$$\psi_n^k(\vec{x}) = J_r(a)J_s^{2s+1,0}(b)(1-b)^r \quad r,s = 0,1,\ldots,M \text{ s.t. } r+s = n \leq M, \tag{2.2.10}$$

where

$$a = 2\frac{1+x}{1-y} - 1, b = y. \tag{2.2.11}$$

We get the local polynomial space of dimension

$$N_{base} = \frac{(M+1)\cdot(M+2)\cdot\ldots\cdot(M+d)}{d!}. \tag{2.2.12}$$

In both 2D cases the mapping between n and r and s (i.e., ordering of basis functions)

$$n = indx(r,s), \tag{2.2.13}$$

with its reverses

$$r = indx_1(n),\ s = indx_2(n), \tag{2.2.14}$$

is theoretically arbitrary. In practice we choose it so that the basis functions are ordered lexically i.e.

$$n \leq m \Leftrightarrow indx_1(n) + indx_2(n) \leq indx_1(m) + indx_2(m). \tag{2.2.15}$$

The used mapping is probably best expressed using a procedural programming language, see Listing 3.2. In the whole computational domain Ω_h the solution can be then thought of as being a member of the direct sum of local spaces

$$Le_{\Omega_h}^M = \bigoplus_{T^k \in \Omega_h} V_{T^k}, \tag{2.2.16}$$

which is the finite dimensional subspace of the broken Sobolev space defined in (2.2.2), that is $Le_{\Omega_h}^M \subset W^{1,2}(\Omega_h)$, Le stands for Legendre, it has dimension

$$N_{dof} = \dim(Le_{\Omega_h}^M) = N \cdot N_{base}, \tag{2.2.17}$$

where subscript dof stands for degrees of freedom (DOFs). This is the local basis commonly used in literature [15], [4], however there are also other usable bases, which must not be orthogonal or polynomial [25]. We will always use the full basis of the functions, however the implementation contains mechanism to omit some of them for testing purposes.

2.3 Spatial discretization

We can now start formulating the discretization in space domain. To discretize equation (2.1.1) in finite elements manner we first devise the weak formulation of the problem. First we choose the unknown p and an arbitrary test function w to be from $C^1(\Omega_h)$ and multiply equation (2.1.1) by w to get

$$\nabla \cdot \vec{f}(p) \cdot w(\vec{x}) - \nabla \cdot (D\nabla p) \cdot w(\vec{x}) = g \cdot w(\vec{x}). \tag{2.3.1}$$

After integrating over the domain Ω we get

$$\int_\Omega \nabla \cdot \vec{f}(p) \cdot w(\vec{x}) - \int_\Omega \nabla \cdot (D\nabla p) \cdot w(\vec{x}) = \int_\Omega g \cdot w(\vec{x}). \tag{2.3.2}$$

This holds for every Cauchy sequence of functions p_n, w_n and using Lebesgue dominated convergence theorem we can formulate the problem on closure of $C^1(\Omega_h)$ i.e. for $p \in W^{1,2}(\Omega_h)$ and $w \in W^{1,2}(\Omega_h)$, obtaining the equation (2.3.2) in the form (we drop independent variables t and $\vec{x}$ notations for brevity)

$$\sum_{k=0}^{N} \left(\int_{T^k} \nabla \cdot \vec{f}(p) \cdot w - \int_{T^k} \nabla \cdot (D\nabla p) \cdot w \right) = \sum_{k=0}^{N} \left(\int_{T^k} g \cdot w \right). \tag{2.3.3}$$

Having arrived to the "broken" integral formulation of the equation we will now focus on discretization of individual terms within mesh elements.

2.3.1 Hyperbolic term discretization

Using the Green's theorem on the first integral term in (2.3.3) we get

$$\int_{T^k} \nabla \cdot \vec{f}(p) \cdot w = \int_{\partial T^k} \vec{n}\vec{f}(p)w - \int_{T^k} \vec{f}(p) \cdot \nabla w, \tag{2.3.4}$$

where $\vec{n}$ is the normal vector to the boundary ∂T^k of T^k. The approximation of the value of $\vec{f}$ on the boundary of the element plays the key role in the discretization using DG FE methods. The issue is that the approximate solution is discontinuous across the boundary of an element and two values are actually present, p_{in} inner to the element and p_{out} outer, coming from its neighbor across a particular part of the boundary. Since we deal with 2D elements with a polygonal boundary, the integral over the boundary can be expressed as sum of integrals over line segments F_i^k, $i = 0, 1, 2$ for triangular meshes or $i = 0, 1, 2, 3$ for quadrilateral meshes, forming the boundary:

$$\sum_{i=0}^{N_f} \int_{F_i^k} \vec{n}_f \vec{f}(p)w. \tag{2.3.5}$$

If we denote $T^{k(i)}$ the element sharing the line segment F_i^k with the element T^k, the value p_{out} corresponds to the approximation in this element, i.e., $p_{out} = p_h^{k(i)}$. For simplicity of notation we continue using integral over the whole boundary of T^k implicitly assuming that p_{out} changes as described above. To approximate the unknown value of $\vec{f}(p)$ we will use the approximate flux $\vec{f}^*(p_{in}, p_{out})$ obtaining the first term on the right-hand side in the form

$$\int_{\partial T^k} \vec{n}\vec{f}(p)w = \int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot w. \tag{2.3.6}$$

In our setting we will use the so-called local Lax-Friedrichs flux exclusively, although there are many other possible fluxes, for examples see [19, 9], their later implementation should be straightforward (more in Section 3.5.1). The Lax-Friedrichs flux as given in [15] is of the form

$$\vec{f}^*(p_{in}, p_{out}) = \frac{\vec{f}(p_{in}) + \vec{f}(p_{out})}{2} + (1 - \alpha)\vec{n}\frac{C}{2}(p_{in} - p_{out}), \tag{2.3.7}$$

where $\alpha \in [0,1]$ is a parameter adjusting the nature of the flux, $\alpha = 0$ for purely upwind scheme, $\alpha = 1$ for central scheme, and

$$C = \max_{p\in[p_{in}:p_{out}]} \left| n_1 \frac{\partial f_1}{\partial p} + n_2 \frac{\partial f_2}{\partial p} \right| = \max_{p\in[p_{in}:p_{out}]} \left| \vec{n} \cdot \frac{d\vec{f}}{dp}(p) \right|, \tag{2.3.8}$$

where $[p_{in} : p_{out}]$ denotes the closed interval

$$\big[\min_{\partial T^k}(\min(p_{in}, p_{out})), \max_{\partial T^k}(\max(p_{in}, p_{out}))\big].$$

Note that in the linear case, where $\vec{f}(p) = \vec{a}p$ and $\frac{d\vec{f}}{dp}(p) = \vec{a}$, C reduces to

$$C = |\vec{n}\vec{a}|. \tag{2.3.9}$$

In this formulation C constitutes the upper bound on wave speed at the boundary interface. In order to simplify notation we denote "jump" in the quantity p across the boundary

$$[p] = p_{in} - p_{out}, \tag{2.3.10}$$

and the average of p across the boundary

$$\langle p \rangle = \frac{p_{in} + p_{out}}{2}. \tag{2.3.11}$$

Using this notation we can write

$$\vec{f}^*(p_{in}, p_{out}) = \langle p \rangle + (1-\alpha)\vec{n}\frac{C}{2}[p]. \tag{2.3.12}$$

After approximating p and w on each element T^k as linear combinations of basis functions as in (2.2.3):

$$p(t, \vec{x}) \approx \sum_{i=0}^{N_{base}-1} P_i^k \psi_i(\vec{x}), \tag{2.3.13}$$

$$w(t, \vec{x}) \approx \sum_{j=0}^{N_{base}-1} W_j^k \psi_j(\vec{x}), \tag{2.3.14}$$

and substituting (2.3.6) to (2.3.4) we arrive to

$$\int_{T^k} \nabla \cdot \vec{f}(p) \cdot w \approx \int_{T^k} \vec{f}\Big(\sum_{i=0}^{N_{base}-1} P_i^k \psi_i\Big) \cdot \nabla\Big(\sum_{j=0}^{N_{base}-1} W_j^k \psi_j\Big) - \int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \sum_{j=0}^{N_{base}-1} W_j^k \psi_j. \tag{2.3.15}$$

Since this approximation holds for every test function $w \in Le^M_{\Omega_h}$ we can successively choose $W_j^k = 1 \; \forall \, j, k$. Using summation notation for clarity, we can then write terms on the right-hand side of (2.3.15) as

$$a^C_{\text{hyp}}(\mathbf{p}) = \int_{T^k} \vec{f}(P_i^k \psi_i) \cdot \nabla \psi_j, \tag{2.3.16}$$

$$a^F_{\text{hyp}}(\mathbf{p}) = \int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \psi_j. \tag{2.3.17}$$

where $\mathbf{p}$ denotes vector of unknowns P_i^k grouped by element as follows

$$\mathbf{p} = \left(P_0^0, P_1^0, P_2^0, \ldots, P_{N_{base}-1}^0, \ \ldots \ , P_0^N, P_1^N, P_2^N, \ldots, P_{N_{base}-1}^N\right). \tag{2.3.18}$$

Note that in (2.3.17) we do not include the minus sign in front of the flux term. In the linear case the term defined in (2.3.17) can be rewritten as

$$a_{\text{adv}}^C(\mathbf{p}) = \int_{T^k} \vec{a} P_i^k \psi_i \cdot \nabla \psi_j. \tag{2.3.19}$$

This finalizes discretization of the general hyperbolic term $\nabla \cdot \vec{f}(p) \cdot w$, the two terms – the integral over element T^k (often called the stiffness term) and the integral over its surface ∂T^k – are implemented in *SfePy* as `ScalarDotMGradScalarTerm` and `AdvectDGFluxTerm` in the special case $\vec{f} = \vec{a}p$ and as `NonlinScalarDotGradTerm` and `NonlinearHyperDGFluxTerm` in the general case. See Section 3.5.1 in Chapter 3 for details on the implementation.

2.3.2 Elliptic term discretization

To discretize the elliptic diffusion term

$$\int_{T^k} \nabla \cdot (D \nabla p) w,$$

we use the Green's theorem as well, obtaining

$$\int_{T^k} \nabla \cdot (D \nabla p) \cdot w = \int_{\partial T^k} D(\nabla p \cdot \vec{n}) w - \int_{T^k} D \nabla p \nabla w. \tag{2.3.20}$$

On the boundary ∂T^k we define, using notation from (2.3.11),

$$\nabla p = \frac{\nabla p_{in} + \nabla p_{out}}{2} = \langle \nabla p \rangle. \tag{2.3.21}$$

Substituting (2.3.21) to (2.3.20) and using the fact that the test function w vanishes outside the element we get the so-called incomplete scheme

$$\int_{\partial T^k} D \langle \nabla p \rangle \cdot \vec{n}[w] - \int_{T^k} D \nabla p \nabla w. \tag{2.3.22}$$

Due to regularity of p the $[p(t, \cdot)] = 0$ holds [19, p. 14] and the term:

$$\int_{\partial T^k} D \langle \nabla w \rangle \cdot \vec{n}[p] \tag{2.3.23}$$

vanishes. We can then create symmetric resp. non-symmetric schemes by adding the term (2.3.23) with "+" resp. "−" sign [19, p. 14]. This "symmetrizes" the discretization with respect to p and w, which is advantageous in theoretical analysis [12, p. 39]. However neither of these schemes is stable and we need to compensate for the discontinuities of the p across element boundaries by adding an interior penalty term [19, 3, 12]

$$\nu \int_{\partial T^k} C_w \cdot \frac{Ord^2}{d(\partial T^k)} [p][w], \tag{2.3.24}$$

where the constant ν captures properties of the diffusion tensor D. In case $D = \varepsilon$, $\varepsilon > 0$ we set $\nu = \varepsilon$. C_w is a parameter at our disposal used to fine tune the penalty term. See examples 3, 4, 5, 7 and 8 in Chapter 4 for effects of different choices of C_w. And finally $d(\partial T^k)$ is the volume of the boundary of T^k. To simplify notation we denote

$$\sigma = \nu C_w \cdot \frac{Ord^2}{d(\partial T^k)}. \tag{2.3.25}$$

We further proceed as for the hyperbolic term. By replacing p and w by their finite dimensional approximations (2.3.13) and (2.3.14) and using the fact that the test function ψ_i^k vanishes outside element T^k and hence

$$\langle \psi_i^k \rangle = \frac{\psi_i^k}{2}, \tag{2.3.26}$$

$$[\psi_i^k] = \psi_i^k, \tag{2.3.27}$$

holds, we obtain individual terms needed to discretize the diffusion term in forms

$$\int_{\partial T^k} D\langle \nabla p \rangle \cdot \vec{n}[w] \approx a_{\text{diff}}^L(\mathbf{p}) = \int_{\partial T^k} D\langle P_i^k \nabla \psi_i \rangle \cdot \vec{n}[\psi_j] = \int_{\partial T^k} D\langle P_i^k \nabla \psi_i \rangle \cdot \vec{n}\psi_j, \tag{2.3.28}$$

$$\int_{\partial T^k} D\langle \nabla w \rangle \cdot \vec{n}[p] \approx a_{\text{diff}}^R(\mathbf{p}) = \int_{\partial T^k} D\langle \nabla \psi_j \rangle \cdot \vec{n}[P_i^k \psi_i] = \int_{\partial T^k} D\frac{\nabla \psi_j}{2} \cdot \vec{n}[P_i^k \psi_i], \tag{2.3.29}$$

$$\int_{T^k} D\nabla p \nabla w \approx a_{\text{diff}}^C(\mathbf{p}) = \int_{T^k} D\nabla P_i^k \psi_i \nabla \psi_j, \tag{2.3.30}$$

$$\int_{\partial T^k} \sigma[p][w] \approx a_{\text{diff}}^P(\mathbf{p}) = \int_{\partial T^k} \sigma[P_i^k \psi_i][\psi_j] = \int_{\partial T^k} \sigma[P_i^k \psi_i]\psi_j. \tag{2.3.31}$$

Using these we construct three variants of the discontinuous Galerkin method used for elliptic problems: Symmetric Interior Penalty Galerkin method (SIPG)

$$\int_{T^k} \nabla \cdot (D\nabla p) w \approx -a_{\text{diff}}^C(\mathbf{p}) + a_{\text{diff}}^L(\mathbf{p}) + a_{\text{diff}}^R(\mathbf{p}) - a_{\text{diff}}^P(\mathbf{p}), \tag{2.3.32}$$

Non-symmetric Interior Penalty Galerkin method (NIPG)

$$\int_{T^k} \nabla \cdot (D\nabla p) w \approx -a_{\text{diff}}^C(\mathbf{p}) + a_{\text{diff}}^L(\mathbf{p}) - a_{\text{diff}}^R(\mathbf{p}) - a_{\text{diff}}^P(\mathbf{p}), \tag{2.3.33}$$

Incomplete Interior Penalty Galerkin method (IIPG)

$$\int_{T^k} \nabla \cdot (D\nabla p) w \approx -a_{\text{diff}}^C(\mathbf{p}) + a_{\text{diff}}^L(\mathbf{p}) - a_{\text{diff}}^P(\mathbf{p}). \tag{2.3.34}$$

2.3.3 Source term discretization

The discretization of the source term is analogous in DG FEM as in continuous FEM. In the term

$$\int_{\Omega_h} g \cdot w \tag{2.3.35}$$

we take the test function from the broken Legendre space $Le_{\Omega_h}^M$ obtaining

$$\sum_{k=0}^{N} \left(\int_{T^k} g \cdot w \right). \tag{2.3.36}$$

After substituting (2.3.14) and successively choosing $W_j^k = 1 \; \forall \, j, k$ we get

$$\sum_{k=0}^{N-1} \sum_{j=0}^{N_{base}-1} \left(\int_{T^k} g \cdot \psi_j \right). \tag{2.3.37}$$

Hence for the element T^k and test function ψ_j the source term coefficient is

$$b_{source} = \int_{T^k} g \cdot \psi_j. \tag{2.3.38}$$

2.4 Temporal discretization

In discretizing the transient equation (2.1.3) we use the discretization of terms devised for the stationary equation, with the important difference that the discretization coefficients in (2.3.13) now depend on time, that is

$$P_i^k = P_i^k(t). \tag{2.4.1}$$

By applying spatial discretization to the transient term we obtain

$$\int_{T^k} \frac{\partial p}{\partial t}(t) w \approx \frac{\mathrm{d}P_i^k}{\mathrm{d}t}(t) \int_{T^k} \psi_i \psi_j. \tag{2.4.2}$$

Using this discretization in (2.1.3) we obtain the system of ordinary differential equations for unknown coefficients $P_i^k(t)$ in variable t

$$\mathbf{M}\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t}(t) + \mathcal{L}(\mathbf{p}(t)) = 0, \tag{2.4.3}$$

where $\mathcal{L}$ is composed from discretized terms derived in previous sections and $\mathbf{M}$ is a matrix composed of blocks $M^{k,l},\ k, l = 0, \ldots, N$ of the form

$$(M^{k,l})_{ij} = (\psi_i^k, \psi_j^l)_{T^k}. \tag{2.4.4}$$

Since the basis functions are orthogonal with respect to the scalar product (2.2.6), the individual blocks are diagonal and since basis functions ψ_i^k vanish outside of the element T^k, the matrix $\mathbf{M}$ is diagonal

$$\mathbf{M} = \begin{pmatrix} M^{0,0} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & M^{1,1} & & & & \\ \vdots & 0 & \ddots & & 0 & \vdots \\ \vdots & \vdots & & M^{k,k} & & \\ \vdots & \vdots & 0 & & \ddots & 0 \\ 0 & 0 & \cdots & \cdots & 0 & M^{N,N} \end{pmatrix}. \tag{2.4.5}$$

Thanks to this the inverse of $\mathbf{M}$ is trivial and we can rewrite (2.4.3) as

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t}(t) + \mathbf{M}^{-1}\mathcal{L}(\mathbf{p}(t)) = 0, \tag{2.4.6}$$

denoting

$$\bar{\mathcal{L}} = \mathbf{M}^{-1}\mathcal{L}, \tag{2.4.7}$$

we can write (2.4.3) in the form

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t}(t) + \bar{\mathcal{L}}(\mathbf{p}(t)) = 0. \tag{2.4.8}$$

There is plethora of different schemes for evolving the equation (2.4.8). We will only present the basic forward Euler scheme and the so called total variations diminishing Runge-Kutta scheme of the 3rd order.

Forward Euler scheme In the forward Euler scheme we approximate the time derivative using the forward difference, i.e.,

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t}(t) \approx \frac{\mathbf{p}^{(n+1)} - \mathbf{p}^{(n)}}{\Delta t}, \tag{2.4.9}$$

where n denotes the current time step. Substituting into (2.4.8) yields

$$\frac{\mathbf{p}^{(n+1)} - \mathbf{p}^{(n)}}{\Delta t} + \bar{\mathcal{L}}(\mathbf{p}^{(n)}, t^{(n)}) = 0 \tag{2.4.10}$$

and after rearranging to obtain an explicit equation for $\mathbf{p}^{(n+1)}$ we get

$$\mathbf{p}^{(n+1)} = \mathbf{p}^{(n)} - \Delta t \bar{\mathcal{L}}(\mathbf{p}^{(n)}, t^{(n)}). \tag{2.4.11}$$

The forward Euler scheme is first order in time. We use it to define the so-called total variation diminishing property of $\bar{\mathcal{L}}$, that is the total variation of the numerical solution in one dimension

$$TV(p) = \sum_k |p_{k+1} - p_k|, \tag{2.4.12}$$

where k ranges over subsequent 1D mesh elements, does not increase in time, i.e.,

$$TV(\mathbf{p}^{n+1}) \leq TV(\mathbf{p}^n), \tag{2.4.13}$$

under update by the forward Euler scheme. This motivates usage of the following TVD Runge-Kutta method [14, p. 73].

TVD Runge-Kutta 3rd order scheme The third order total variations diminishing Runge-Kutta scheme [14] is a three step scheme that maintains the TVD property while achieving the 3rd order accuracy in time.

$$\begin{aligned} \mathbf{p}^{(1)} &= \mathbf{p}^{(n)} - \Delta t \bar{\mathcal{L}}(\mathbf{p}^{(n)}), \\ \mathbf{p}^{(2)} &= \frac{3}{4}\mathbf{p}^{(n)} + \frac{1}{4}\mathbf{p}^{(1)} - \frac{1}{4}\Delta t \bar{\mathcal{L}}(\mathbf{p}^{(1)}), \\ \mathbf{p}^{(n+1)} &= \frac{1}{3}\mathbf{p}^{(n)} + \frac{2}{3}\mathbf{p}^{(2)} - \frac{2}{3}\Delta t \bar{\mathcal{L}}(\mathbf{p}^{(2)}). \end{aligned} \tag{2.4.14}$$

Hyperbolic term stability requirement Use of explicit time stepping solvers poses strict upper bounds on the size of time step. For purely advection problems the Courant-Friedrichs-Lewy condition adjusted for high order approximations mandates [5, p. 5]

$$\Delta t \leq C_{\mathrm{CFL}} \frac{h}{\|\vec{a}\|} \cdot \frac{1}{2M+1}, \tag{2.4.15}$$

where $0 < C_{\mathrm{CFL}} \leq 1$ is an adjustable parameter of order 1.

Elliptic term stability requirement For problems including diffusion the CFL condition (2.4.15) is often overridden by [15]

$$\Delta t \leq C_{\mathrm{DIFF}} \frac{h^2}{D}, \tag{2.4.16}$$

where $0 < C_{\mathrm{DIFF}} \leq 1$ is again an adjustable parameter of order 1.

2.5 Initial condition discretization

The initial condition

$$p_0(\vec{x}),$$

is discretized in a straight-forward manner as an orthogonal projection into the finite dimensional space $Le^M_{\Omega_h}$ on the domain Ω_h. That is by solving

$$(P_i^k)^0 \int_{T^k} \psi_i \psi_j = \int_{T^k} p_0(\vec{x}) \psi_j, \tag{2.5.1}$$

for $(P_i^k)^0$. We use the mass matrix notation and the fact that it is diagonal and get

$$(P_i^k)^0 = \frac{1}{(\psi_i^k, \psi_i^k)_{T^k}} \int_{T^k} p_0(\vec{x}) \psi_i. \tag{2.5.2}$$

2.6 Boundary conditions

Unlike it is common in literature we postponed treatment of the boundary conditions (BCs) until now. The reason is to keep the theoretical discussion closely tied with the implementation. This allows us to demonstrate how the method works, hopefully providing the reader with enough information and understanding to modify it. In our implementation treatment of boundary conditions is separated from the terms implementation, i.e., terms do not have any information about boundary conditions, they are merely passed data, which already satisfy BCs. In expressions

$$\langle p \rangle = \frac{p_{in} + p_{out}}{2} \tag{2.6.1}$$

and

$$[p] = p_{in} - p_{out}, \tag{2.6.2}$$

the missing outer value p_{out} in boundary elements is substituted by

- the value of the Dirichlet boundary condition, or
- the value in the corresponding neighbor cell where the periodic boundary conditions are defined,

whenever there is no direct neighbor. In implementations this is ensured in terms themselves by getting corresponding values from `DGField` through method `get_both_facet_base_vals`.

2.7 Limiters

In high order DG FEM, oscillations, which can significantly decrease the quality of solution, occur even when the Courant-Friedrichs-Lewy condition (2.4.15) is met. To combat this a limiter needs to be used — in the following section we present moment limiters for 1D and 2D problems.

2.7.1 Moment limiter

The moment limiter by Krivodonova [18] leverages the idea that coefficients for higher-order basis functions in hierarchically ordered Legendre basis represent derivatives of lower-order data and uses this to limit the derivative of order i in a given cell using derivatives of order $i-1$ in neighboring cells. This kind of limiter, unlike others, does not reduce the solution to the first-order accuracy. Unfortunately this kind of limiting is so far available only in one-dimensional problems and in two dimensional problems with tensor product meshes.

One-dimensional limiting

We limit the solution in each cell T^k

$$p_h^k(t,\vec{x}) = \sum_{i=0}^{N_{base}} P_i^k \psi_i(\vec{x}) \tag{2.7.1}$$

by limiting its coefficients P_i^k, starting with the coefficients of the highest order, i.e., $i = N_{base}$ we subsequently replace P_i^k with

$$\tilde{P}_i^k = \text{minmod}\left(P_i^k, \alpha_i(P_{i-1}^{k+1} - P_{i-1}^k), \alpha_i(P_{i-1}^k - P_{i-1}^{k-1})\right), \tag{2.7.2}$$

stopping when $P_i^k = \tilde{P}_i^k$. In the definition of limiter (2.7.2), minmod is a function of three variables

$$\text{minmod}(a,b,c) = \begin{cases} \text{sign}(a)\min(|a|,|b|,|c|) & \text{if } \text{sign}(a) = \text{sign}(b) = \text{sign}(c) \\ 0 & \text{otherwise} \end{cases}. \tag{2.7.3}$$

and α_i is the limiting coefficient dependent on the order. Krivodonova [18] proposes to take α_i from range

$$\frac{1}{2(2n-1)} \leq \alpha_n \leq 1. \tag{2.7.4}$$

Choosing α_i outside this region results in either a loss of accuracy or a numerical instability [18, p. 882]. The lower bound of the interval corresponds to the strictest limiting, whereas $\alpha_i = 1$ is the mildest limiter possible [18, p. 882] and we set it as the value of α. The one-dimensional limiter is implemented in `dg.limiters.MomentLimiter1D`, see Section 3.6 for details.

Two-dimensional limiting

In this section we describe extension of the moment limiter to regular tensor-product meshes in two dimensions. We limit the solution coefficients in individual cells much like in 1D, but this time we have to take into account derivatives in four directions and to introduce some new notation:

$$\begin{aligned}\tilde{P}_{r,s}^{k,m} = \text{minmod}\big(&P_{r,s}^{k,m}, \alpha_s(P_{r,s-1}^{k,m+1} - P_{r,s-1}^{k,m}), \alpha_s(P_{r,s-1}^{k,m} - P_{r,s-1}^{k,m-1}), \\ &\alpha_r(P_{r-1,s}^{k+1,m} - P_{r-1,s}^{k+1,m}), \alpha_r(P_{r-1,s}^{k-1,m} - P_{r-1,s}^{k,m})\big),\end{aligned} \tag{2.7.5}$$

where $P_{r,s}^{k,m}$ denotes the coefficient of the basis function ψ_n of the form

$$\psi_n = L^r(x)L^s(y)$$

i.e.,

$$n = indx(r, s).$$

Further we leveraged the fact that we assume the mesh to be regular uniform Cartesian grid and introduced the new notation for indexing mesh elements as $T^{k,m}$ where k ranges over rows and m over columns. The region of stability for α_n is different due to normalization of basis functions

$$\frac{1}{2\sqrt{4n^2-1}} \leq \alpha_n \leq \sqrt{\frac{2n-1}{2n+1}}, \tag{2.7.6}$$

we choose upper bound as α again. The two-dimensional limiter is implemented in the class `dg.limiters.MomentLimiter2D`, see Section 3.6.

Chapter 3

Discontinous Galerkin Method implementation

In this chapter we explore in detail *SfePy* package application interface (API) as well as its inner workings in order to explain the implementation details of the method. We will show several usage examples and hopefully provide enough information for users to use the method effectively and even modify it.

3.1 Problem specification

Before we delve into inner workings of *SfePy* numerical code lets introduce the so-called declarative problem specification format. The format relies on Python dictionaries, see Listing 3.1 for model problem specification for the 2D Laplace equation on the domain $[0, 1] \times [0, 1]$ simplified from Example 3. It contains dictionaries declaring components of the problem like regions in a geometric domain, a field governing the used FE method, state, and test variables, boundary conditions, material constants, and functions, etc. Detailed and more general treatment of the format can be found in [8] here we focus on the specification of a problem to be solved using DG FEM.

Listing 3.1: Problem specification file `example_dg_laplace.py`.

```
regions = {'Omega'     : 'all',  ❶
'left' : ('vertices in x == 0', 'edge'),
'right': ('vertices in x == 1', 'edge'),
'top' : ('vertices in y == 1', 'edge'),
'bottom': ('vertices in y == 0', 'edge')}
fields = {'f': ('real', 'scalar', 'Omega', ❷
                str(approx_order) + 'd', 'DG', 'legendre')}

variables = {'p': ('unknown field', 'f', 0, 1),
             'v': ('test field', 'f', 'p')} ❸

def analytic_sol(coors):
  x_1, x_2 = coors[..., 0], coors[..., 1]
  res = 1/2*x_1**2 - 1/2*x_2**2 - a*x_1 + b*x_2 + c
  return res

@local_register_function
def bcs(ts, coors, bc, problem): ❹
  x_1, x_2 = coors[..., 0], coors[..., 1]
```

```
  res = nm.zeros(x_1.shape)
  if bc.diff == 0:
      res[:] = analytic_sol(coors)
  elif bc.diff == 1:
      res = nm.stack((x_1 - a, -x_2 + b), axis=-2)
  return res

dgebcs = { ❺
  'p_left' : ('left', {'p.all': "bcs", 'grad.p.all': "bcs"}),
  'p_right' : ('right', {'p.all': "bcs", 'grad.p.all': "bcs"}),
  'p_bottom' : ('bottom', {'p.all': "bcs", 'grad.p.all': "bcs"}),
  'p_top' : ('top', {'p.all': "bcs", 'grad.p.all': "bcs"})}

materials = {'D': ({'val': [diffcoef], '.Cw': cw},)} ❻
integrals = {'i': 2 * approx_order}

equations = {'the_equation': ❼
  "dw_laplace.i.Omega(D.val, v, p) "
  " - dw_dg_diffusion_flux.i.Omega(D.val, p, v)"
  " - dw_dg_diffusion_flux.i.Omega(D.val, v, p)"
  " + dw_dg_interior_penalty.i.Omega(D.val, D.Cw, v, p)"
  "= 0"}

solvers = {'ls': ('ls.auto_direct', {}),❽
           'newton': ('nls.newton', {})}
options = {'nls'           : 'newton', ❾
           'ls'            : 'ls',
           'output_format' : 'msh'
           'format_variant': 'gmsh-dg'}
```

---

❶ `regions` dictionary specifies different regions used in boundary conditions specification, `Omega` region is required for setting up fields, `'edge'` regions are needed for BCs.

❷ `fields` determine discretization spaces of variables and are defined using tuple of strings. `(data type, number of components, region name, approximation order, field type, polyspace)` here the field `'f'` is a field for the discretization of a real scalar variable in the region `"Omega"` using the discontinuous Galerkin method of the order `approx_order` in the space of Legendre polynomials.

❸ Here `'p'` is an unknown state variable we are solving for and `'v'` is a test variable, they are both discretized using the field `'f'` defined above.

❹ The function `bcs` is called during solution, it is supposed to produce values and derivatives of boundary conditions.

❹ The `dgebcs` dictionary sets up boundary conditions specifically for DG FE methods, it creates map between boundary regions and, variables and values of functions that determine boundary conditions.

❻ In materials dictionary we specify the diffusion coefficient D, the dot notation `'.Cw'` causes material not to be broad-casted to quadrature points, which is convenient for constants parameterizing terms like C_w or α in (2.3.25) resp. (2.3.7).

❼ The equation to solve is composed of terms derived in Chapter 2.

❽ Linear and non-linear solvers to use, *SfePy* supports various solvers including `mumps` [2].

❾ Various options, `'output_format': 'msh'` and `'format_variant': 'gmsh'` ensure output in format suitable for post-processing using `Gmsh` [21](`http://gmsh.info/`).

3.2 *SfePy* architecture

Components in the problem specification file are parsed into various Python objects brought together in the `Problem` object, the most important are:

- `Equation` – representing the equation to be solved,
- `EssentialBC` – representing Dirichlet boundary conditions,
- `PeriodicBC` – representing periodic boundary conditions,
- `InitialCondition` – representing the initial condition, in case of transient problems,
- `TimeSteppingSolver` – specifying the time discretization scheme, in case of transient problems.

The `Equation` object is built by combining `Term` objects: these represent individual integral terms that are evaluated in the course of solving a problem. Due to the interpreted nature of CPython in which *SfePy* is mainly run and which is generally too slow for high-performance numerical computation due to overhead from the interpreter, *SfePy* relies on various approaches to speed up the computation. In general, it uses fast vectorized operations provided by NumPy and SciPy [23]. C and Cython are used in places where vectorization is not possible, or is too difficult or unreadable [8]. Our implementation relies on NumPy vectorization, especially the `einsum` function (mode details below). Terms keep references to other objects:

- `Variable` – representing state and test variables,
- `Material` – representing various material constants or functions,
- `Integral` – representing Gauss quadrature rules.

The `Variable` object in turn contains reference to the `Field` object that manages the chosen discretization – finite dimensional space represented by a `PolySpace` object and provides methods needed to work with it along with the computational domain (the `Domain` object) which stores geometry including `Region` objects used in the definition of `EssentialBC` and `PeriodicBC` and the equations.

3.3 DG method components

Having laid out the structure of the *SfePy* problem and objects needed to create it and work with it we now present classes needed to implement the DG FEM. Following the architecture of *SfePy*, the DG FE method implementation comprises of:

- `DGField`,
- `LegendrePolySpace` and its subclasses,
- `LegendreTensorProductPolySpace` and
- `LegendreSimplexPolySpace`;

DG specific terms as summarized in the top portion of Table 3.3.1;

Table 3.3.1: Table of terms used in DG method.

Class	Name	Symbol	Expression
`AdvectionDGFluxTerm`	`"dw_dg_advect_laxfrie_flux"`	$a^F_{\text{adv}}(\mathbf{p})$	$\int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \psi_j$
`NonlinearHyperbolicDGFluxTerm`	`"dw_dg_nonlinear_laxfrie_flux"`	$a^F_{\text{hyp}}(\mathbf{p})$	$\int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \psi_j$
`NonlinearScalarDotGradTerm`	`"dw_ns_dot_grad_s"`	$a^C_{\text{hyp}}(\mathbf{p})$	$\int_{T^k} \vec{f}(P^k_i \psi_i) \cdot \nabla \psi_j$
`DiffusionDGFluxTerm`	`"dw_dg_diffusion_flux"`	$a^R_{\text{diff}}(\mathbf{p})$ $a^L_{\text{diff}}(\mathbf{p})$	$\int_{\partial T^k} D \frac{\nabla \psi_j}{2} \cdot \vec{n}[P^k_i \psi_i]$ $\int_{\partial T^k} D \langle P^k_i \nabla \psi_i \rangle \cdot \vec{n} \psi_j$
`DiffusionInteriorPenaltyTerm`	`"dw_dg_interior_penalty"`	$a^P_{\text{diff}}(\mathbf{p})$	$\int_{\partial T^k} \sigma [P^k_i \psi_i] \psi_j$
`ScalarDotMGradScalarTerm`	`"dw_s_dot_mgrad_s"`	$a^C_{\text{adv}}(\mathbf{p})$	$\int_{T^k} \vec{a} P^k_i \psi_i \cdot \nabla \psi_j$
`LaplaceTerm`	`"dw_laplace"`	$a^C_{\text{diff}}(\mathbf{p})$	$\int_{T^k} D \nabla P^k_i \psi_i \nabla \psi_j$
`DotProductVolumeTerm`	`"dw_volume_dot"`	–	$\frac{\mathrm{d}P^k_i}{\mathrm{d}t}(t) \int_{T^k} \psi_i \psi_j$

DG specific boundary conditions:

- `DGEssentialBC`,
- `DGPeriodicBC`;

and multistage time-stepping solvers:

- abstract base class `DGMultiStageTS` and two solvers used in numerical experiments:
- `EulerStepSolver`,
- `TVDRK3StepSolver`.

Finally limiters were implemented as subclasses of `DGLimiter` abstract class (which has no counterpart in *SfePy*):

- `IdentityLimiter` – provided for convenience to enable easily disabling limiter without changing syntax,
- `MomentLimiter1D` – for 1D problems only,
- `MommentLimiter2D` – only for 2D problems on regular tensor product meshes.

The limiters are used in the problem composition as post-stage hooks passed to time-stepping solvers. For technical reasons we also created the `DGVariable` class in order to bypass the classical FE treatment of boundary conditions, otherwise it is similar to the original *SfePy* `Variable` class and we omit its detailed description.

3.4 DG Field

The `DGField` class inherits from the `Field` base class. This provides it with the basic functionality needed to be used in problem specification. From methods implemented in `DGField`, the most relevant to DG FEM are:

- `get_both_facet_state_vals` – which returns values of state variable on opposing sides of the boundary for each element,
- `get_both_facet_base_vals` – which returns values of basis functions on opposing sides of the boundary for each element,
- `get_facet_neighbor_idx` – which returns indices of cell neighbors for individual facets along with index of the facet within the neighboring cell,

- `get_bc_facet_values` – which provides values of boundary conditions,
- `get_facet_boundary_idx` –
- `get_facet_vols`
- `get_facet_qp`

3.4.1 Legendre polynomial spaces implementation

Legendre polynomial spaces are implemented in two classes `LegendreTensorProductPolySpace` and `LegendreSimplexPolySpace`. Both are derived from the abstract class `LegendrePolySpace` which inherits from *SfePy* `PolySpace`. It implements the method `_eval_base` which is used to get values of basis functions as well as their derivatives. It also contains methods for evaluating Legendre and Jacobi polynomials common to tensor-product and simplex subclasses. These classes are accompanied by the function `get_n_el_nod`, which returns number of basis functions for the given order, dimension and type of basis, and the generator `iter_by_order` (3.2) which generates tuples of r and s in desired hierarchical order. For example, for the approximation order 2 and the tensor-product basis this is: `(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (2, 1), (1, 2), (2, 2)`.

Listing 3.2: Iteration over r and s indicies of basis functions .

```
def iter_by_order(order, dim, extended=False):

  ...

  porder = order + 1
  for k in range(porder):
    for r in range(k + 1):
      yield r, k - r ❶
    if not extended: return ❷
    for s in range(1, porder):
      for r in range(1, porder):
        if r + s <= porder - 1:
          continue
        yield r, s
```

❶ `yield` keyword turns a function into a generator usable in for cycles, for example in Listing 3.6.

❷ `extended` flag distinguishes the simplex basis from tensor-product one which uses more basis functions.

To obtain values of Jacobi polynomials, we used implementations provided by SciPy in the `special` module.

3.5 DG Terms

Besides terms listed in Table 3.3.1 we implemented the abstract class `DGTerm` from which the other terms inherit. Methods `eval_real` and `call_function` implemented in this class manage calling the method named `function`, which each term implements, and returning the results to the evaluation engine of *SfePy*. The `function` method comes from architecture of the terms already present in *SfePy* where it is used to call extensions programmed and optimized using C

programing language. This method is called whenever value of the term is needed either to build residual vector (i.e. right-hand side of an equation) or to get terms contribution to the matrix form of $\mathcal{L}$ (in case of implicit problem). The method returns the residual values corresponding to the individual DOFs and in the matrix mode also the indices to build the sparse matrix representation. To demonstrate how this is implemented we explore `AdvectionDGFluxTerm` and `DiffusionDGFluxTerm`, `DiffusionInteriorPenaltyTerm` is implemented in the same manner, `NonlinearScalarDotGradTerm` was modified from `ScalarDotMGradScalarTerm` already implemented in *SfePy*.

3.5.1 Hyperbolic flux term implementation

`AdvectionDGFluxTerm` corresponds to the discretized term (2.3.17) where $\vec{f}(p) = \vec{a}p$. The part of the `function` capturing computation of cell fluxes can be seen in Listing 3.3 below.

Listing 3.3: Computation of advection cell fluxes.

```
def function(self, out, state, diff_var, field, region, advelo):

  fc_n = field.get_cell_normals_per_facet(region)
  # get maximal wave speeds at facets
  C = nm.abs(nm.einsum("ifk,ik->if", fc_n, advelo))  ❶

  if diff_var is not None: ❷

    nbrhd_idx = field.get_facet_neighbor_idx(region, state.eq_map)
    active_cells, active_facets = nm.where(nbrhd_idx[:, :, 0] >= 0)
    active_nrbhs = nbrhd_idx[active_cells, active_facets, 0]

    in_fc_b, out_fc_b, whs = field.get_both_facet_base_vals(state,
       region)

    inner_diff = nm.einsum("nfk, nfk->nf",  ❸
                           fc_n,
                           advelo[:, None, :]
                           + nm.einsum("nfk, nf->nfk",
                             (1 - self.alpha) * fc_n, C)) / 2.
    outer_diff = nm.einsum("nfk, nfk->nf",
                           fc_n,
                           advelo[:, None, :]
                           - nm.einsum("nfk, nf->nfk",
                           (1 - self.alpha) * fc_n, C)) / 2.

    inner_vals = nm.einsum("nf, ndfq, nbfq, nfq -> ndb",  ❹
                           inner_diff,
                           in_fc_b,
                           in_fc_b,
                           whs)
    outer_vals = nm.einsum("i, idq, ibq, iq -> idb",
                          outer_diff[active_cells, active_facets],
                          in_fc_b[active_cells, :, active_facets],
                          out_fc_b[active_cells, :, active_facets],
                          whs[active_cells, active_facets])

    vals = nm.vstack((inner_vals, outer_vals))
    vals = vals.flatten()
```

```
        # compute positions within matrix
        iels = self._get_nbrhd_dof_indexes(active_cells, active_nrbhs,
            field)

        out = (vals, iels[:, 0], iels[:, 1], state, state)

    else:

        facet_base_vals = field.get_facet_base(base_only=True)
        in_fc_v, out_fc_v, weights =
            field.get_both_facet_state_vals(state, region)

        # reshape facet base to (n_el_nod, n_el_facet, n_qp)
        fc_b = facet_base_vals[:, 0, :, 0, :].T

        fc_v_avg = (in_fc_v + out_fc_v)/2.
        fc_v_jmp = in_fc_v - out_fc_v

        central = nm.einsum("ik,ifq->ifkq", advelo, fc_v_avg)
        upwind = (1 - self.alpha)/2. * nm.einsum("if,ifk,ifq->ifkq",
                                                 C, fc_n, fc_v_jmp)

        cell_fluxes = nm.einsum("ifk,ifkq,dfq,ifq->id",
                                fc_n, central + upwind, fc_b, weights)
        out [0, 0, :, 0] = cell_fluxes

    return out
```

❶ `numpy.einsum` uses the Einstein summation notation for expressing tensor contractions, for details see [10].

❷ The presence of `diff_var` denotes an evaluation in the matrix mode.

❸, ❹ Variables `inner_diff`, `outer_diff` and `inner_vals` and `outer_vals` correspond to the decomposition of the term (2.3.17)

$$\int_{\partial T^k} \vec{n} \cdot \vec{f^*}(p_{in}.p_{out}) \cdot \psi_j \tag{3.5.1}$$

First we substitute the flux $\vec{f^*}$ from(2.3.7), using $\vec{f}(p) = \vec{a}p$ we get

$$\int_{\partial T^k} \vec{n} \left(\frac{\vec{a}p_{in} + \vec{a}p_{out}}{2} + (1-\alpha)\vec{n}\frac{C}{2}(p_{in} - p_{out}) \right) \psi_j \tag{3.5.2}$$

and then expand and split the integral

$$\int_{\partial T^k} \vec{n}\frac{\vec{a}p_{in}}{2}\psi_j + \int_{\partial T^k} \vec{n}\frac{\vec{a}p_{out}}{2}\psi_j + \int_{\partial T^k} \vec{n}(1-\alpha)\vec{n}\frac{C}{2}p_{in}\psi_j - \int_{\partial T^k} \vec{n}(1-\alpha)\vec{n}\frac{C}{2}p_{out}\psi_j. \tag{3.5.3}$$

Rearranging yields

$$\int_{\partial T^k} \vec{n}\frac{\vec{a}p_{in}}{2}\psi_j + \int_{\partial T^k} \vec{n}(1-\alpha)\vec{n}\frac{C}{2}p_{in}\psi_j + \int_{\partial T^k} \vec{n}\frac{\vec{a}p_{out}}{2}\psi_j - \int_{\partial T^k} \vec{n}(1-\alpha)\vec{n}\frac{C}{2}p_{out}\psi_j, \tag{3.5.4}$$

$$\int_{\partial T^k} \frac{1}{2}(\vec{n}\vec{a} + \vec{n}(1-\alpha)\vec{n}C)p_{in}\psi_j + \int_{\partial T^k} \frac{1}{2}(\vec{n}\vec{a} - \vec{n}(1-\alpha)\vec{n}C)p_{out}\psi_j, \tag{3.5.5}$$

after substituting corresponding outer and inner values of p we get the contribution for P_i^k and $P_i^{k(l)}$ in the form

$$\overbrace{\int_{\partial T^k} \underbrace{\frac{1}{2}\vec{n}(\vec{a}+(1-\alpha)\vec{n}C)}_{\texttt{inner\_diff}}\psi_i^k\psi_j}^{\texttt{inner\_vals}} \tag{3.5.6}$$

and

$$\overbrace{\int_{\partial T^k} \underbrace{\frac{1}{2}\vec{n}(\vec{a}-(1-\alpha)\vec{n}C)}_{\texttt{outer\_diff}}\psi_i^{k(l)}\psi_j^k}^{\texttt{outer\_vals}}, \tag{3.5.7}$$

where $k(l)$ denotes the neighboring element sharing face F_l^k like in (2.3.5).

The general hyperbolic term is implemented in the class `NonlinearHyperbolicDGFluxTerm` unlike linear advection term above it does not support evaluation in matrix mode.

3.5.2 Diffusion flux term implementation

Implementation of the diffusion flux terms follows the same course as the implementation of hyperbolic flux terms, with the important difference that `DiffusionDGFluxTerm` implements both terms in (2.3.28) and (2.3.29). This is thanks to two modes in which it can be used in an equation — this has already been demonstrated for the Laplace equation in Listing 3.1 where `"dw_dg_diffusion_flux.i.Omega(D.val, p, v)"` corresponds to $a^R_{\mathrm{diff}}(\mathbf{p})$ and mode `'avg_state'` (❶), and `"dw_dg_diffusion_flux.i.Omega(D.val, v, p)"` corresponds to $a^L_{\mathrm{diff}}(\mathbf{p})$ and mode `'avg_virtual'` (❷). The implementation of residual mode computation is presented in Listing 3.4.

Listing 3.4: Computation of diffusion cell fluxes.

```
if self.mode == 'avg_state': ❶
  avgDdState = (nm.einsum("ikl,ifkq->ifkq",
                          D, inner_facet_state_d) +
                nm.einsum("ikl,ifkq->ifkq",
                          D, outer_facet_state_d)) / 2.
  # outer_facet_base is in DG zero
  # hence the jump is inner value
  jmpBase = inner_facet_base

  cell_fluxes = nm.einsum("ifkq ,ifk,idfq,ifq->id",
                          avgDdState, fc_n, jmpBase, weights)

elif self.mode == 'avg_virtual': ❷
  avgDdbase = (nm.einsum("ikl,idfkq->idfkq",
                         D, inner_facet_base_d)) / 2.

  jmpState = inner_facet_state - outer_facet_state
  cell_fluxes = nm.einsum("idfkq, ifk, ifq , ifq -> id",
                          avgDdbase, fc_n, jmpState, weights)
```

3.6 Limiters implementation

Following design patterns used in *SfePy* and Python in general, the limiters are implemented as classes. The base class providing only the constructor is called `DGLimiter`, its subclasses then

implement the abstract method `__call__` — this makes all limiters callable objects, allowing one to pass them as post-step or post-stage or other hooks to time-stepping solvers. For convenience the identity limiter which does not alter the solution is implemented in the class `IdentityLimiter`.

Moment limiter – 1D

The code listing below shows the implementation of the moment limiter introduced in Section 2.7.1, omitting some details for brevity.

Listing 3.5: Moment limiter for 1D.

```
idx = nm.arange(nm.shape(u[0, 1:-1])[0])

nu = nm.copy(u)
tilu = nm.zeros(u.shape[1:])
for ll in range(self.n_el_nod - 1, 0, -1):
  tilu[idx] = minmod(nu[ll, 1:-1][idx],
                     nu[ll-1, 2:][idx] - nu[ll-1, 1:-1][idx],
                     nu[ll-1, 1:-1][idx] - nu[ll-1, :-2][idx]) ❶

  idx = idx[nm.where(abs(tilu[idx] - nu[ll, 1:-1][idx])
            > MACHINE_EPS)[0]] ❷
  if len(idx) == 0:
    break ❸
  nu[ll, 1:-1][idx] = tilu[idx] ❹
```

❶ Compute the limiting value $\tilde{u}$.

❷ Extract indicies where the limiting value is larger than the current solution.

❸ If none of the coefficients requires limiting we stop.

❸ Replace old values with limited ones.

Moment limiter – 2D

We list the implementation of the 2D limiter for reference in Listing 3.6. The Limiter is implemented according to Section 2.7.

Listing 3.6: Moment limiter for cartesian grid.

```
for ll, (ii, jj) in enumerate(
                     iter_by_order(self.field.approx_order,
                                   2,   # dim
                                   extended=ex)):
  nu[ii, jj, ...] = u[ll] ❶

for ii, jj in reversed(list(
                       iter_by_order(
                           self.field.approx_order, 2,
                           extended=ex))):
  minmod_args = [nu[ii, jj, idx]]
  nbrhs = nbrhd_idx[idx]
  if ii - 1 >= 0:
    alf = nm.sqrt((2 * ii-1) / (2 * ii + 1))
    # right difference in x axis
    dx_r = alf*(nu[ii-1, jj, nbrhs[:, 1]] - nu[ii-1, jj, idx])
```

```
    # left differnce in x axis
    dx_l = alf*(nu[ii-1, jj, idx] - nu[ii-1, jj, nbrhs[:, 3]])
    minmod_args += [dx_r, dx_l]
  if jj - 1 >= 0:
    alf = nm.sqrt((2 * jj - 1) / (2 * jj + 1))
    # right i.e. element "up" difference in y axis
    dy_up = alf*(nu[ii, jj-1, nbrhs[:, 2]] - nu[ii, jj-1,  idx])
    # left i.e. element "down" difference in y axis
    dy_dn = alf*(nu[ii, jj-1,  idx] - nu[ii, jj-1,  nbrhs[:, 0]])
    minmod_args += [dy_up, dy_dn]

  tilu[idx] = minmod_seq(minmod_args)
  idx = idx[nm.where(abs(tilu[idx] - nu[ii, jj, idx]) >
      MACHINE_EPS)[0]]

  if len(idx) == 0:
    break
  nu[ii, jj, idx] = tilu[idx]

resu = nm.zeros(u.shape)
for ll, (ii, jj) in enumerate(
                      iter_by_order(self.field.approx_order,
                                    2,    # dim
                                    extended=ex)):
  resu[ll] = nu[ii, jj] ❷
```

❶ Reshape the solution array for indexing using r and s indicies, effectively removing need for the explicit inverse of index mapping from (2.2.13).

❷ Convert back to the linear index.

3.7 Time-stepping solvers implementation

As demonstrated in Section 2.4, the explicit DG FEM requires explicit time stepping solvers with multiple stages in one time step. These had not been part of the rich collection of time-stepping solvers included in *SfePy*, so two new solvers were implemented: the basic Euler solver, the total-variations diminishing Runge-Kutta of the 3rd order (TVD RK-3). Again following the structure of *SfePy*, they are implemented as subclasses of `TimeSteppingSolver`. The abstract class `DGMultiStageTS` extends the basic `TimeSteppingSolver` with the option to provide pre-stage and post-stage hooks, allowing to apply limiters between stages. The two time-stepping solvers are then implemented in classes `EulerStepSolver` and `TVDRK3StepSolver`.

Chapter 4

Numerical experiments

In this chapter we first introduce PDEs used to study the behavior of DG FE method and provide a short guide to the convergence study setup. Finally, we present the results of convergence studies for various problem setups.

4.1 Example PDEs

In the following examples we will be demonstrating the behavior of the method by solving the following equations:

Transient advection equation in one resp. two dimensions

$$\frac{\partial p}{\partial t} + a\frac{\partial p}{\partial x} = 0, \tag{4.1.1}$$

resp.

$$\frac{\partial p}{\partial t} + \vec{a} \cdot \nabla p = 0. \tag{4.1.2}$$

After applying discretizations devised in Chapter 2 we obtain both equations in the same form

$$\frac{\mathrm{d}P_i^k}{\mathrm{d}t}(t) \int_{T^k} \psi_i \psi_j - \int_{T^k} \vec{a} P_i^k \psi_i \cdot \nabla \psi_j + \int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \psi_j = 0. \tag{4.1.3}$$

The Equation expressed in *SfePy* declarative notation can be found in Listing 4.1 below;

Listing 4.1: Advection equation

```
equations = {'Advection':
  # transient
  "dw_volume_dot.i.Omega(v, p)"

  "- dw_s_dot_mgrad_s.i.Omega(a.val, p[-1], v)" ❶
  "+ dw_dg_advect_laxfrie_flux.i.Omega(a.flux, a.val, v, p[-1])"  ❷
  "= 0"
  }
```

❶ `"p[-1]"` ensures the variable object stores history one step backwards in time facilitating forward nature of time stepping solvers,

❷ `"a.flux"` is an optional material argument representing α coefficient in (2.3.7).

Laplace equation in two dimensions

$$- D \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \right) = 0. \tag{4.1.4}$$

Employing the symmetric discretization of the diffusion term and adding the diffusion penalty term yields the equation in the form

$$\int_{T^k} D \nabla P_i^k \psi_i \nabla \psi_j - \int_{\partial T^k} D \langle P_i^k \nabla \psi_i \rangle \cdot \vec{n} \psi_j - \int_{\partial T^k} D \frac{\nabla \psi_j}{2} \cdot \vec{n} [P_i^k \psi_i] + \nu \int_{\partial T^k} \sigma [P_i^k \psi_i] \psi_j = 0. \tag{4.1.5}$$

The discretization of this equation using *SfePy* terms can be found in Listing 3.1;

Static advection-diffusion equation with a right hand side

$$\frac{\partial p}{\partial x} + \frac{\partial p}{\partial y} - D \cdot \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \right) = g, \tag{4.1.6}$$

i.e.,

$$\vec{a} \cdot \nabla p - D \Delta p = g, \tag{4.1.7}$$

where $\vec{a} = [1, 1]^T$ is the advection velocity, D is the diffusion coefficient and g is a source function. Combining discretizations of the two previous equations we obtain the discretized form

$$- \int_{T^k} \vec{a} P_i^k \psi_i \cdot \nabla \psi_j + \int_{\partial T^k} \vec{n} \cdot \vec{f}^*(p_{in}, p_{out}) \cdot \psi_j + \int_{T^k} D \nabla P_i^k \psi_i \nabla \psi_j - \int_{\partial T^k} D \langle P_i^k \nabla \psi_i \rangle \cdot \vec{n} \psi_j - \int_{\partial T^k} D \frac{\nabla \psi_j}{2} \cdot \vec{n} [P_i^k \psi_i] + \nu \int_{\partial T^k} \sigma [P_i^k \psi_i] \psi_j - \int_{T^k} g \cdot \psi_j = 0. \tag{4.1.8}$$

In *SfePy* declarative notation this equation has the form presented in Listing 4.2.

Listing 4.2: Static advection-diffusion equation

```
equations = {'adv_diff' :
    # advection
    "- dw_s_dot_mgrad_s.i.Omega(a.val, p, v)"
    "+ dw_dg_advect_laxfrie_flux.i.Omega(a.flux, a.val, v, p)"
    # diffusion
    "+ dw_laplace.i.Omega(D.val, v, p) "
    "- dw_dg_diffusion_flux.i.Omega(D.val, p, v)"
    "- dw_dg_diffusion_flux.i.Omega(D.val, v, p)"
    # penalty
    "+ dw_dg_interior_penalty.i.Omega(D.val, D.cw, v, p)"
    # source
    "- dw_volume_lvf.i.Omega(g.val, v)"
    "= 0"
    }
```

Transient viscous Burgers' equation in one resp. two dimensions

$$\frac{\partial p}{\partial t} + \frac{1}{2}\frac{\partial p^2}{\partial x} - D \cdot \frac{\partial^2 p}{\partial x^2} = g, \tag{4.1.9}$$

resp.

$$\frac{\partial p}{\partial t} + \frac{1}{2}\left(\frac{\partial p^2}{\partial x} + \frac{\partial p^2}{\partial y}\right) - D \cdot \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}\right) = g, \tag{4.1.10}$$

i.e.,

$$\frac{\partial p}{\partial t} + \nabla \cdot \vec{f}(p) - D\Delta p = g. \tag{4.1.11}$$

with $\vec{f}(p) = \frac{1}{2}[p^2, p^2]^T = \frac{1}{2}\vec{a}p^2$. Discretizing using all the terms derived before we get the same form for both 1D and 2D

$$\begin{aligned}\frac{\mathrm{d}P_i^k}{\mathrm{d}t}(t)\int_{T^k}\psi_i\psi_j - \int_{T^k}\vec{f}(P_i^k\psi_i)\cdot\nabla\psi_j + \int_{\partial T^k}\vec{n}\cdot\vec{f^*}(p_{in},p_{out})\cdot\psi_j \\ + \int_{T^k} D\nabla P_i^k\psi_i\nabla\psi_j - \int_{\partial T^k} D\langle P_i^k\nabla\psi_i\rangle\cdot\vec{n}\psi_j - \int_{\partial T^k} D\frac{\nabla\psi_j}{2}\cdot\vec{n}[P_i^k\psi_i] \\ + \nu\int_{\partial T^k}\sigma[P_i^k\psi_i]\psi_j - \int_{T^k} g\cdot\psi_j = 0.\end{aligned} \tag{4.1.12}$$

In *SfePy* declarative notation this equation has the form presented in Listing 4.3.

Listing 4.3: Viscous Burgers' equation

```
burg_velo = nm.array([1., 1.])

def f(p):
    return .5*burg_velo * p[..., None] ** 2

def f_d(p):
    return burg_velo * p[..., None]

equations = {'burgers':
    # transient
    "dw_volume_dot.i.Omega(v, p)"
    #  non-linear hyperbolic terms
    "- dw_ns_dot_grad_s.i.Omega(f, f_d, p[-1], v)" ❶
    "+ dw_dg_nonlinear_laxfrie_flux.i.Omega(f, f_d, v, p[-1])" ❷
    #  diffusion
    "+ dw_laplace.i.Omega(D.val, v, p[-1])"
    "- dw_dg_diffusion_flux.i.Omega(D.val, p[-1], v)"
    "- dw_dg_diffusion_flux.i.Omega(D.val, v, p[-1])"
    # penalty
    "+ dw_dg_interior_penalty.i.Omega(D.val, D.Cw, v, p[-1])"
    # source
    "- dw_volume_lvf.i.Omega(g.val, v)"
    " = 0"
    }
```

❶, ❷ Nonlinear terms require as parameters the function and its derivative.

4.2 Examples

Measuring convergence We define convergence rate r as is common in literature

$$r = \frac{\log\left(\frac{\|p - p_{h_n}\|_{L^2}}{\|p - p_{h_{n-1}}\|_{L^2}}\right)}{\log\left(\frac{h_n^d}{h_{n-1}^d}\right)}. \tag{4.2.1}$$

Inspired by [19] we present plots depicting average convergence rate over several mesh refinements, this might not be an ideal measure of the method behavior, nevertheless it still provides us with a convenient indicator. Accompanied with plots of L^2 error it allows us to reason about the method over several varying parameters, notably it reveals the relationship between the diffusion coefficient and the penalty term in examples including the diffusion terms.

In Example 1 we explore behavior of DG FE method for the 1D pure advection, the time dependent problem with and without limiting, in Example 2 we do the same for the 2D problem although this time we omit the convergence study in favor of exploring effectiveness of different approximation orders while keeping the number of DOFs constant. Examples 3, 4, 5 and 6 demonstrate importance of diffusion penalty terms. Final two examples 7 and 8 show usage of the method on the Burgers' equation. All the test problems studied further are specified using the declarative approach introduced in 4, codes can be found in [26] (https://zenodo.org/record/3947773).

Example 1 (Advection 1D)**.** In $\Omega = [0, 1]$ we will solve the equation (4.1.1). We use two initial conditions $u(0, x)$ to obtain two different solutions:

$$u_{smooth} = \begin{cases} g(x), & 0.1 < x < .3 \\ 0, & \text{elsewhere} \end{cases}, \tag{4.2.2}$$

where

$$g(\xi) = \exp\left(\frac{1}{10(\xi - 0.2)^2 - 1} + 1\right), \tag{4.2.3}$$

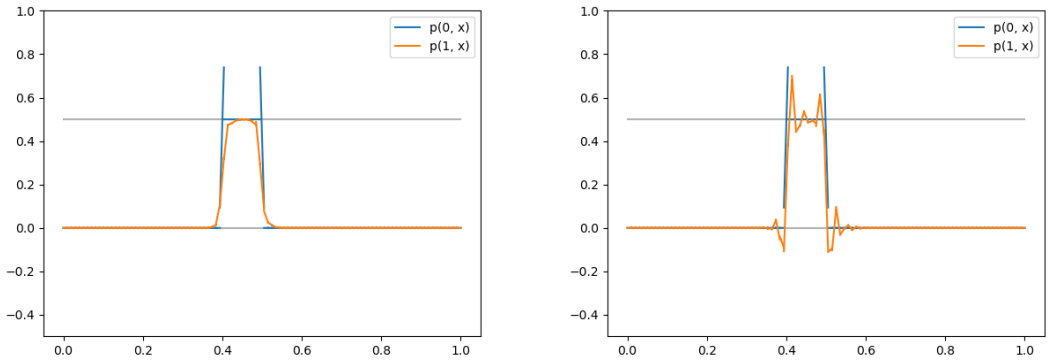
and

$$u_{step} = \begin{cases} \frac{1}{2}, & 0.1 < x < .3 \\ 0, & \text{elsewhere} \end{cases}. \tag{4.2.4}$$

The periodic boundary condition is prescribed at $x = 0$ and $x = 1$ — this results in the solution at time $t = 1$ to be the same as the initial condition, i.e.

$$u(1, x) = u(0, x). \tag{4.2.5}$$

We then compare $u(1, x)$ with $u(0, x)$ to test the convergence, see Figure 2 and 3. For the smooth initial condition the limiting increases the error of the solution due to artificial diffusion, higher order methods are capable of counteracting this effect though. For the discontinuous initial condition the limiting significantly improves the behavior of the method by removing oscillations and basically enabling use of high order methods, which suffer from them the most. This effect is illustrated in Figure 1. The resulting errors are still significant as the limiting introduces prominent smoothing. Note that for both u_{smooth} a u_{step} with and without limiting the convergence rate of the method is impacted by using the 3rd order TVD Runge-Kutta time-stepping solver. This behavior is consistent with that reported by Krivodonova [18].



(a) Limit: True

(b) Limit: False

Figure 1: Example 1. Solution for u_{step} for CFL coefficient $C_{CFL} = 0.1$, order $M = 4$, on uniform mesh with 100 elements.

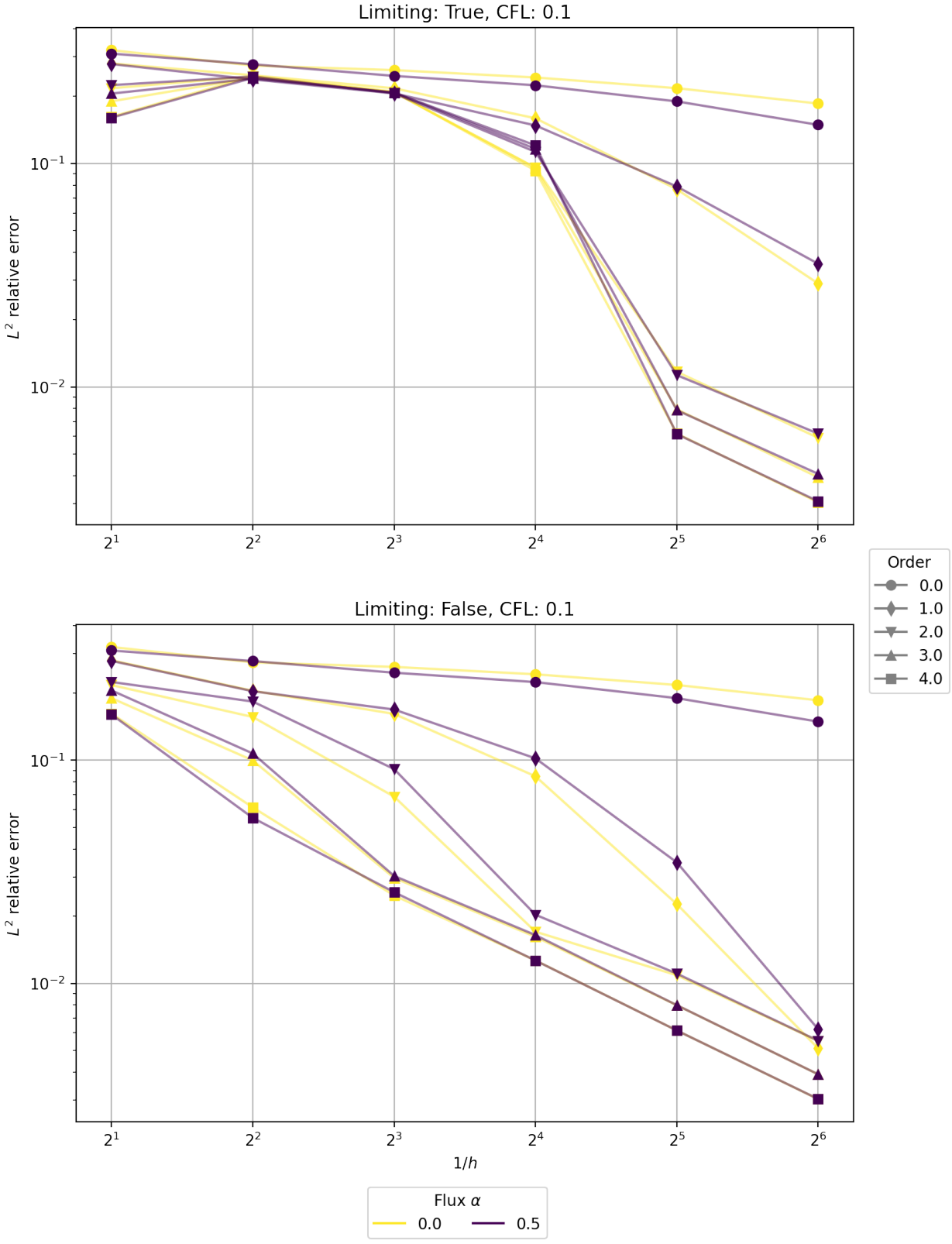


Figure 2: Example 1. Relative errors for smooth initial condition u_{smooth}.

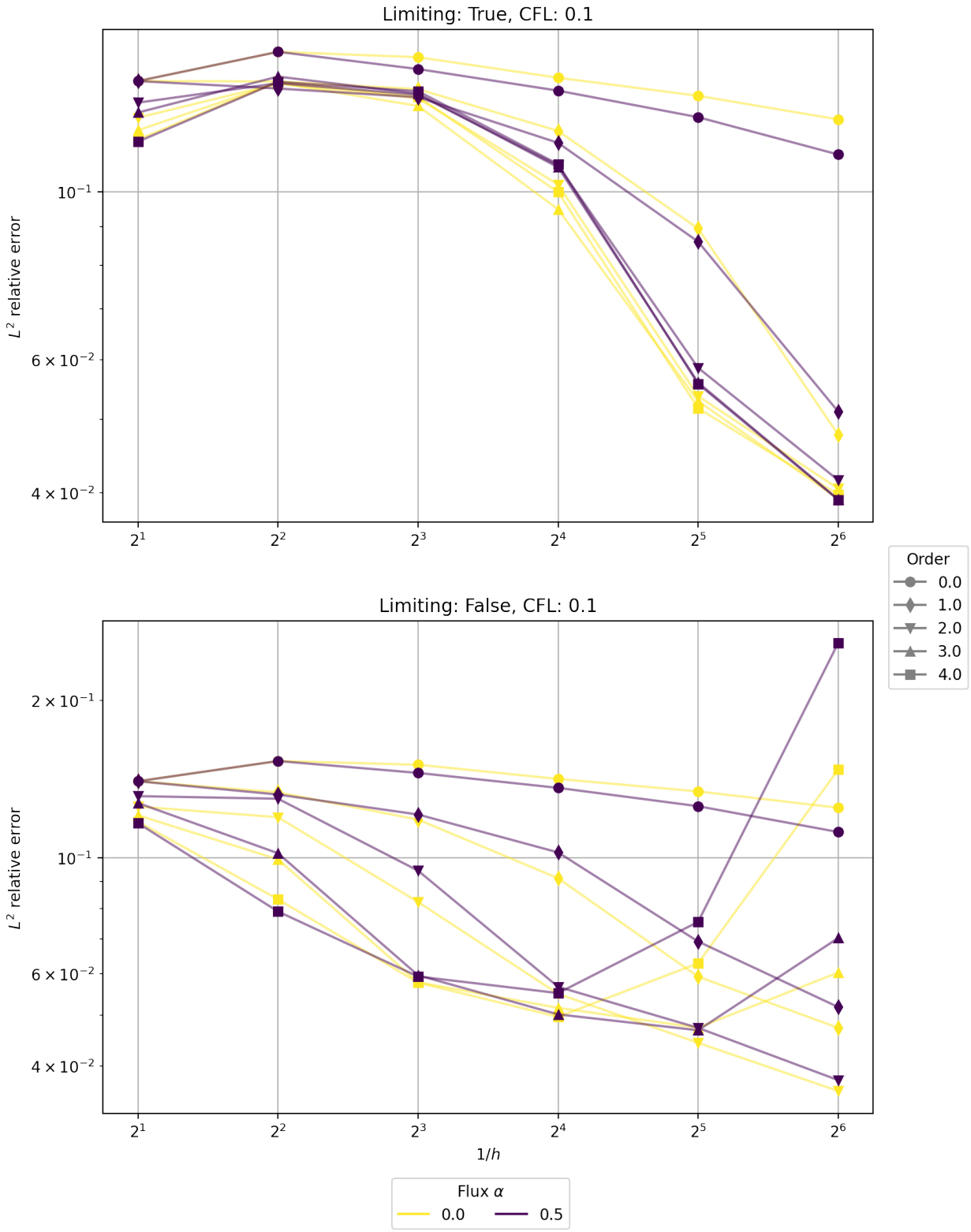


Figure 3: Example 1. Relative errors for discontinuous initial condition u_{step}.

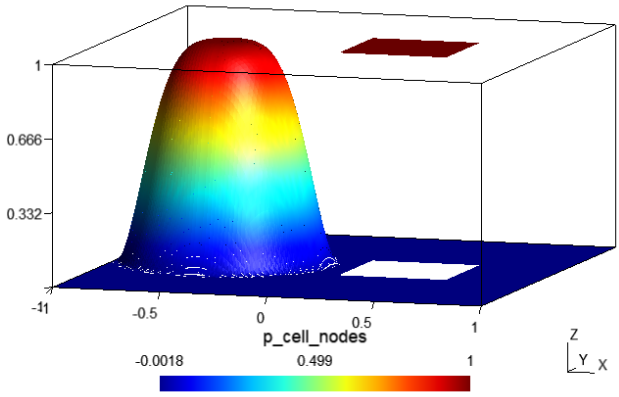
Example 2 (Advection 2D)**.** In $\Omega = [-1, 1]^2$ we will solve the equation (4.1.1). We choose the same initial conditions $u(0, x)$ as in [18]

$$u(x,0) = \begin{cases} \cos^2(2\pi r), & r \leq 0.25, \\ 1, & 0.1 \leq x \leq 0.6 \text{ and } -0.25 \leq y \leq 0.25, \\ 0, & \text{elsewhere}, \end{cases} \tag{4.2.6}$$

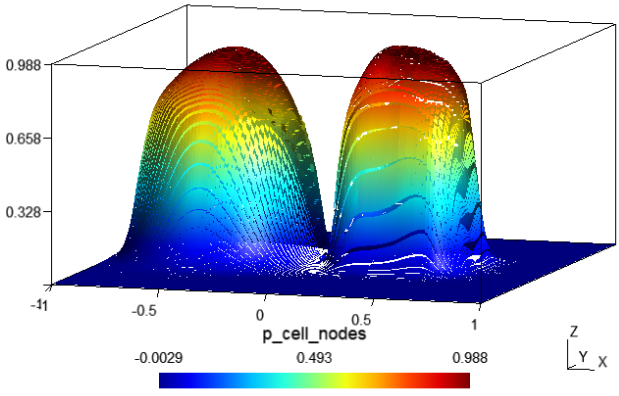
where $r = (x + 0.5)^2 + y$. We choose the velocity $\vec{a}$ dependent on space coordinates:

$$\vec{a} = (2\pi x, -2\pi y),$$

so that the initial data rotate about the origin, revolving fully for every integer value of t. In Figure 4 we can see comparison of the initial state and the state after one revolution. In Table 4.2.1 and Figure 5 we present errors and contours of the solution for different orders, we choose meshes so that the number of DOFs remains the same (57600) for each order. Table 4.2.1 clearly shows that higher orders are not beneficial. And the trade off between refining a mesh and increasing the order seems to favor mesh refining.



(a) $t = 0$

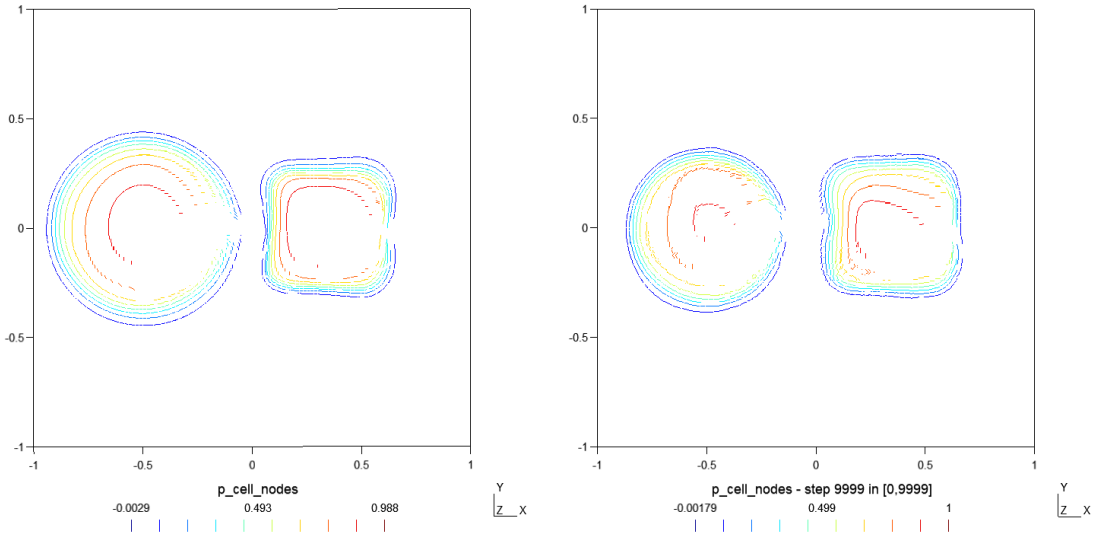


(b) $t = 1$

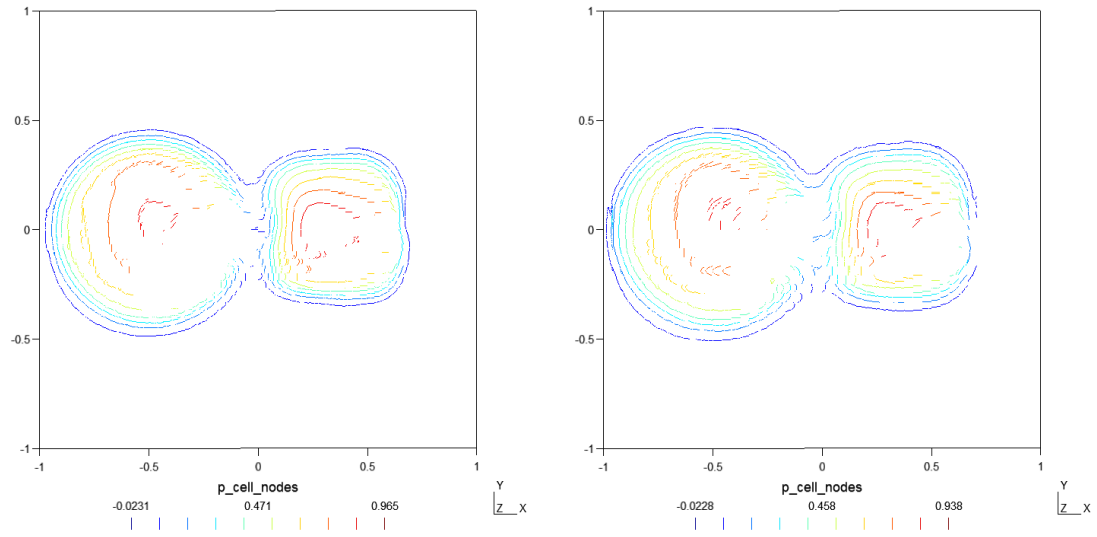
Figure 4: Example 2. Approximation of initial condition (top) and solution after one revolution (bottom) obtained using first order approximation.

Table 4.2.1: Example 2. Errors in L^2 norm for different orders.

Order M	#Cells	N_{base}	Error	Initial error
1	14400	4	0.1577	0.0004
2	6400	9	0.1963	0.0001
3	3600	16	0.2297	0.0648
4	2304	25	0.2619	0.0787



(a) $M = 1$ (b) $M = 2$



(c) $M = 3$ (d) $M = 4$

Figure 5: Example 2. Contours of the solution for different orders, number of DOFs is kept constant.

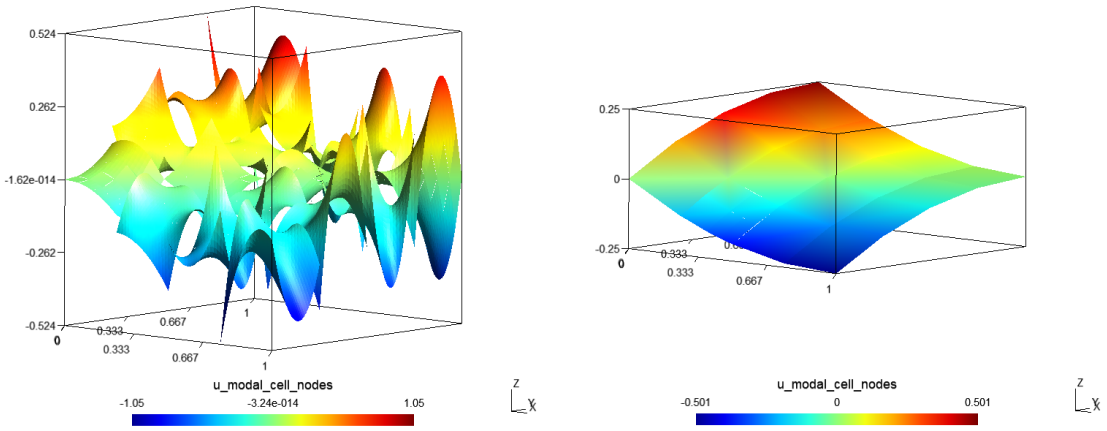
Example 3 (Diffusion 2D)**.** Inspired by [16, problem 8.4 (3), p. 150], in $\Omega = [0,1]^2$ we will solve the Laplace equation (4.1.4). We setup boundary conditions in such a way that the exact solution u_{exact} is polynomial

$$u_{exact} = \frac{1}{2}x^2 - \frac{1}{2}y^2 - ax + by + c. \tag{4.2.7}$$

We set boundary conditions to match the analytical solution as follows

$$\begin{aligned} u_x(0,y) &= -a, \quad u_x(a,y) = 0, \\ u_y(x,0) &= b, \quad\quad u_y(x,b) = 0. \end{aligned} \tag{4.2.8}$$

In our setting we chose $a = 1$, $b = 1$, $c = 0$. Different values of the coefficient C_w in the penalty term yield different convergence behavior as demonstrated in Figures 7 and 8. Figure 6 shows the importance of penalty term for stabilizing the method. Figure 7 may suggest that high order methods do not meet the expected convergence rate, they however still attain the lowest error as illustrated in Figure 8. This is due to the polynomial solution which can be approximated very accurately even on a coarse mesh and refining does not provide much benefit especially for the high order approximations.



(a) $C_w = 1$ (b) $C_w = 10$

Figure 6: Example 3. Solution for different values of C_w, $D = 0.001$, $M = 2$, uniform quadrilateral mesh 4×4 elements. The visualization was scaled down by factor 0.5 in vertical axis.

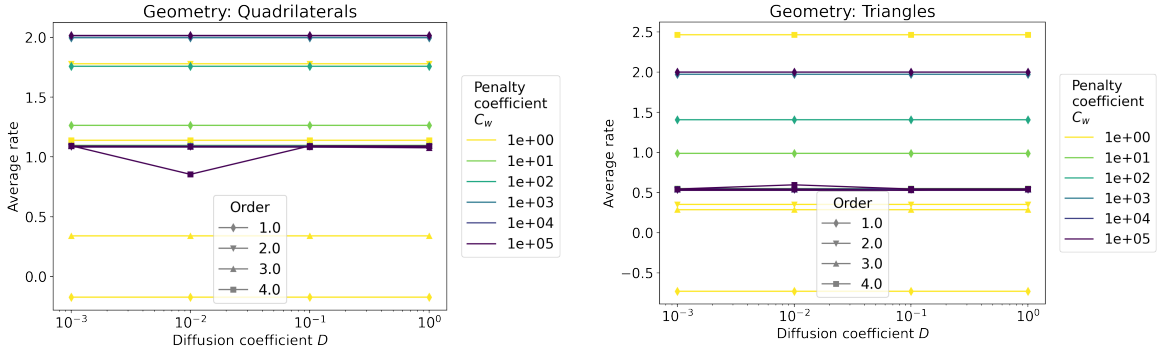


Figure 7: Example 3. Average convergence rates for different choices of C_w for quadrilaterals (left) and triangles (right).

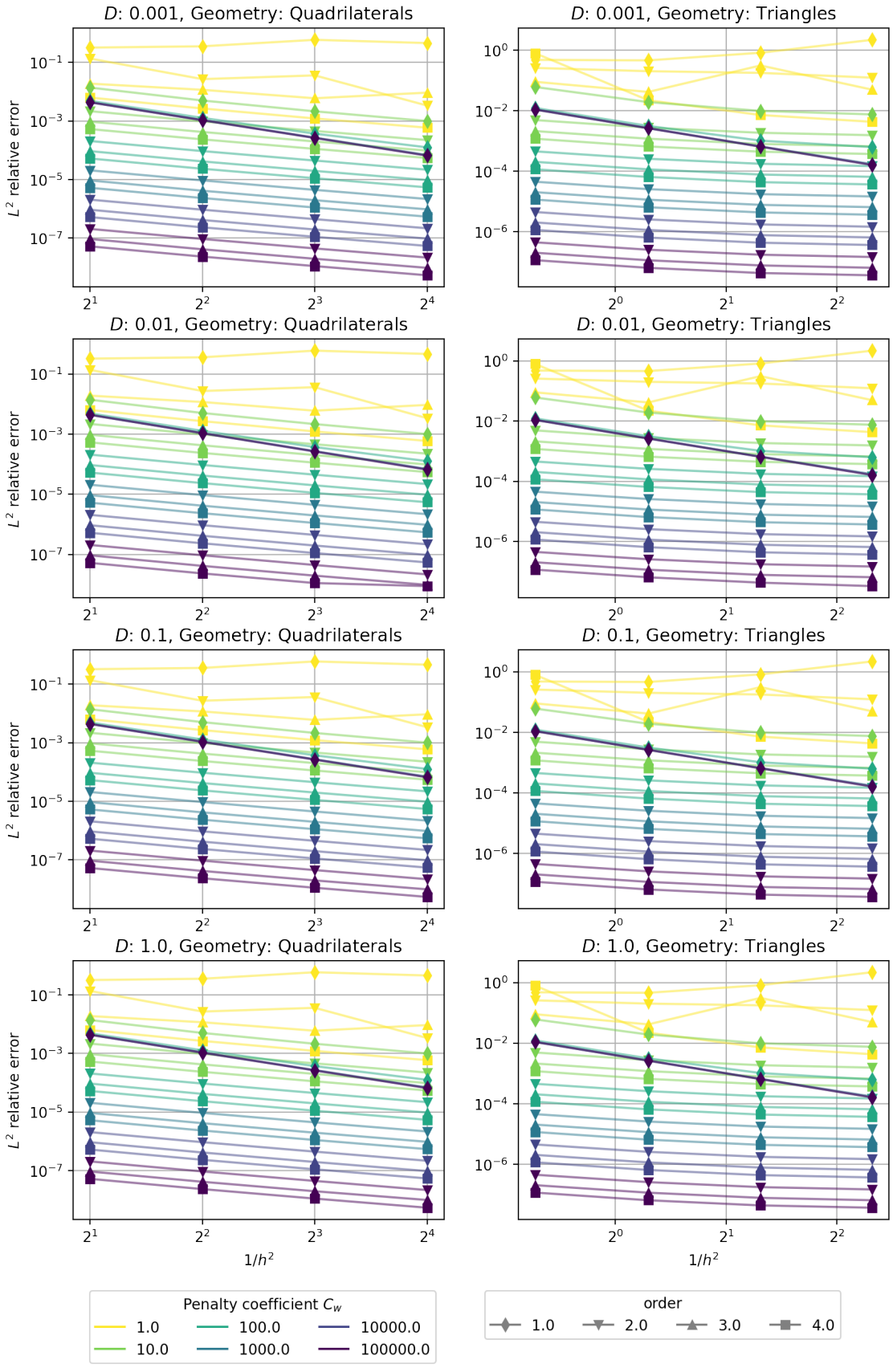


Figure 8: Example 3. Relative errors for different choice of C_w for quadrilaterals (left) and triangles (right).

Example 4 (Advection-diffusion 2D)**.** Based on Example 1 from [3], we will solve the equation (4.1.6) in $\Omega = [0,1]^2$. We set up the boundary conditions and source function g in such a way that the exact solution u_{exact} is

$$u_{exact}(x,y) = -\left(y^2 - y\right)\sin\left(2\,\pi x\right). \tag{4.2.9}$$

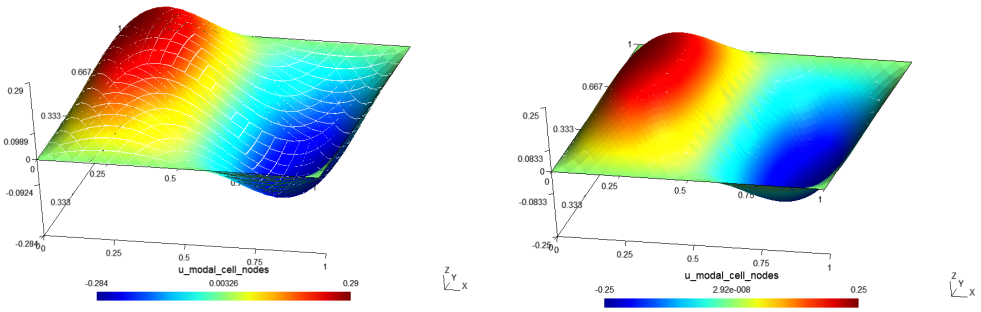
Solving for g yields

$$g = \; -2\,\pi\left(y^2-y\right)\cos\left(2\,\pi x\right) - 2\left(2\,\pi^2\left(y^2-y\right)\sin\left(2\,\pi x\right) - D\sin\left(2\,\pi x\right)\right) - \left(2\,y-1\right)\sin\left(2\,\pi x\right). \tag{4.2.10}$$

Matching boundary conditions are

$$u(x) = 0, \quad \nabla u(x) = [-2\pi(y^2-y)\cos(2\pi x), -(2y-1)\sin(2\pi x)]^T, \quad x \in \partial\Omega. \tag{4.2.11}$$

Different values of the coefficient C_w in the penalty term then yield different convergence behavior as demonstrated in Figures 10 and 11. Both figures illustrate the "gluing" effect of the penalty term which increases with C_w and counteracts discontinuities between elements which are the main source of error in this example. In Figure 9 this effect is clearly visible in numerical solutions. With the growing C_w the convergence behavior of the method improves, with the exception of the 0th order approximation for which it has no effect as expected. Antonietti et al. [3] report convergence for order 1 and 2, these are in accord with ours.



(a) $C_w = 1$ (b) $C_w = 10^5$

Figure 9: Example 4. Solutions on quadrilateral mesh for $D = 1$ and for different values of C_w.

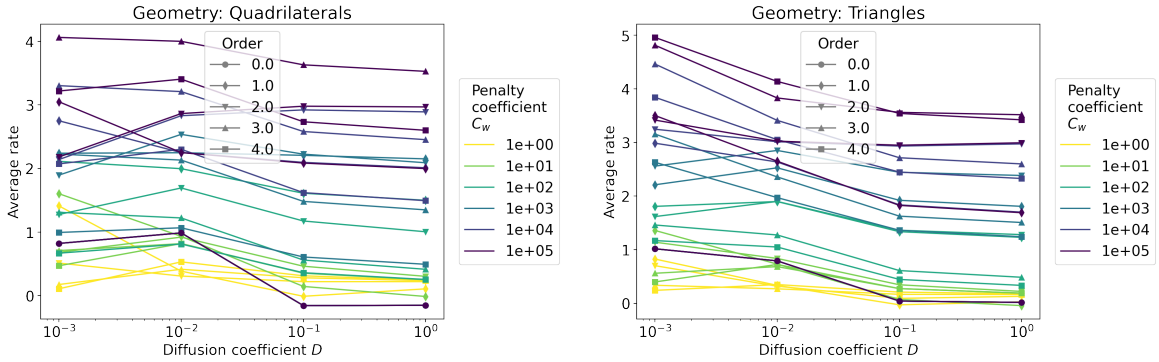


Figure 10: Example 4. Average convergence rates for different choice of C_w for quadrilaterals (left) and triangles (right).

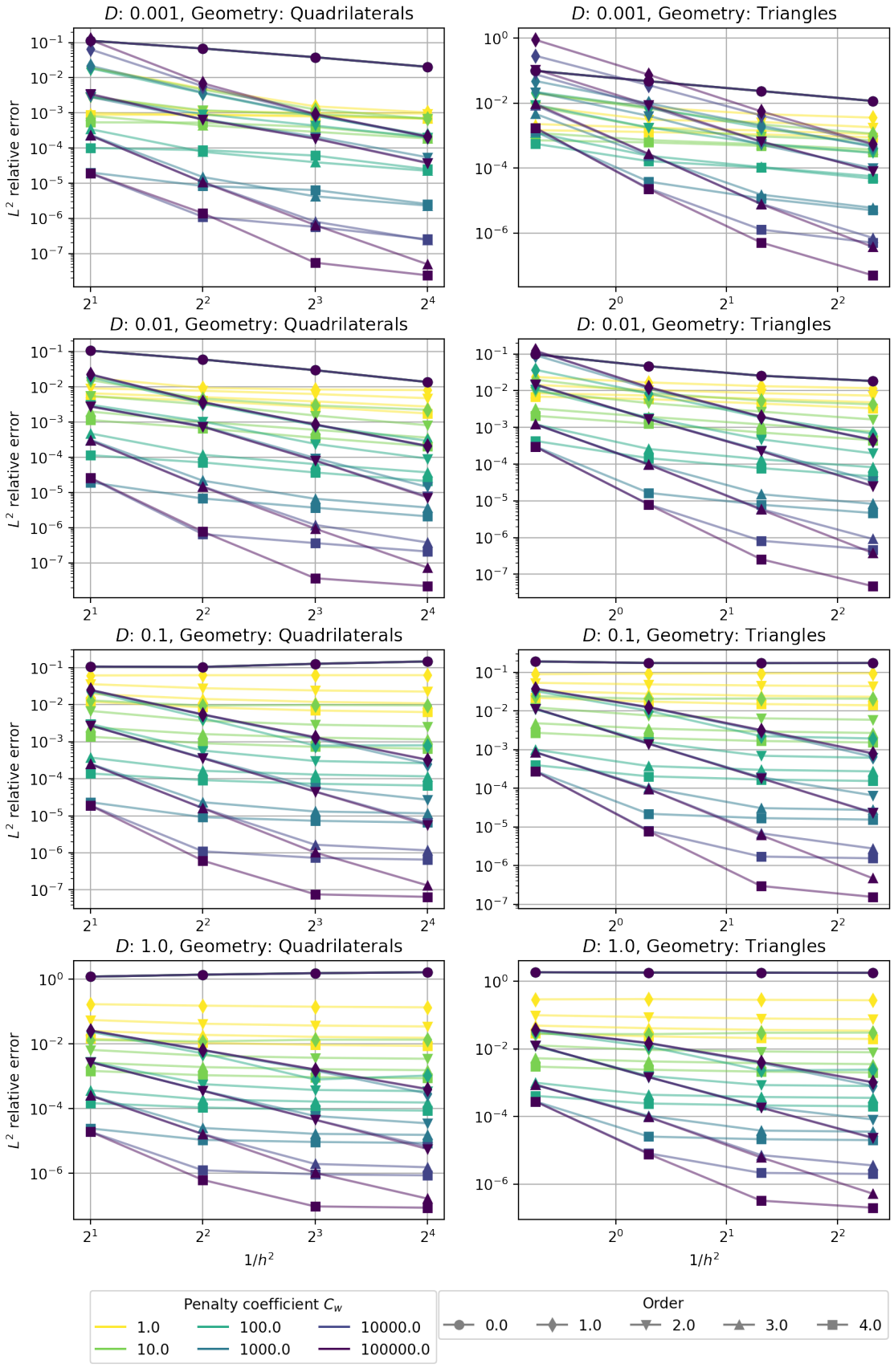
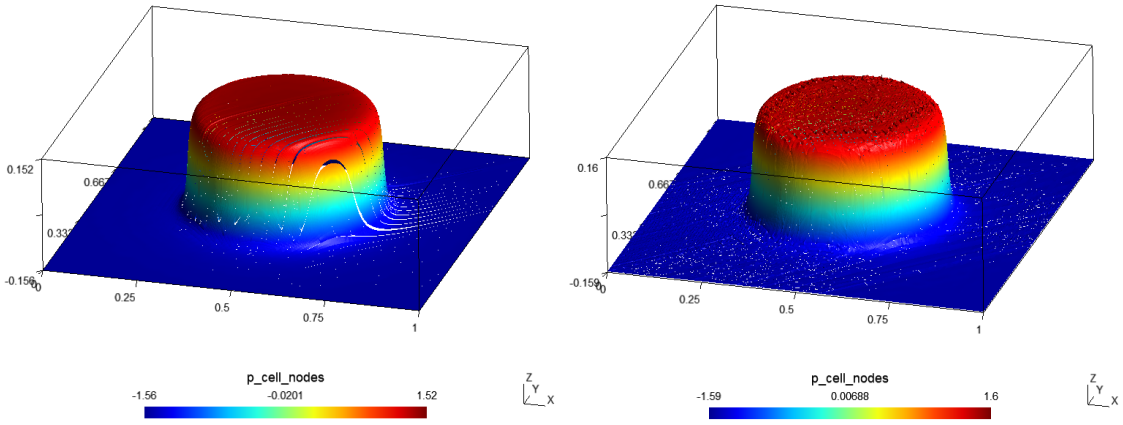


Figure 11: Example 4. Relative errors for different choices of C_w for quadrilaterals (left) and triangles (right).

Example 5 (Advection-diffusion 2D)**.** Based on Example 2 [3], in $\Omega = [0,1]^2$ we will again solve the equation (4.1.6) We set up the boundary condition and source function in such a way that the exact solution u_{exact} is

$$u_{exact} = -\arctan\left(\frac{4\,(2\,x-1)^2 + 4\,(2\,y-1)^2 - 1}{16\,\sqrt{D}}\right). \tag{4.2.12}$$

We omit analytical forms of g and boundary conditions for brevity, they can be found in the code. Different values of the coefficient C_w in the penalty term yield different convergence behavior as demonstrated in Figures 13 and 14. For very low values of the diffusion coefficient D the solution develops into a cylinder. In this state high values of C_w are detrimental as they prevent steep edges and cause artifacts in the solution, for illustration compare Figure 12a and Figure 12b: in (a) the relative error of the approximate solution significantly less then in (b) despite visible discontinuities. Antonietti et al. [3] report convergence for order 1 and 2, these are in accord with ours.



(a) $C_w = 1$ (b) $C_w = 10^5$

Figure 12: Example 5. Solutions for $D = 10^{-5}$, on triangular mesh with 4096 elements, 4th order approximation. The visualization was scaled down by factor 0.1 in vertical axis.

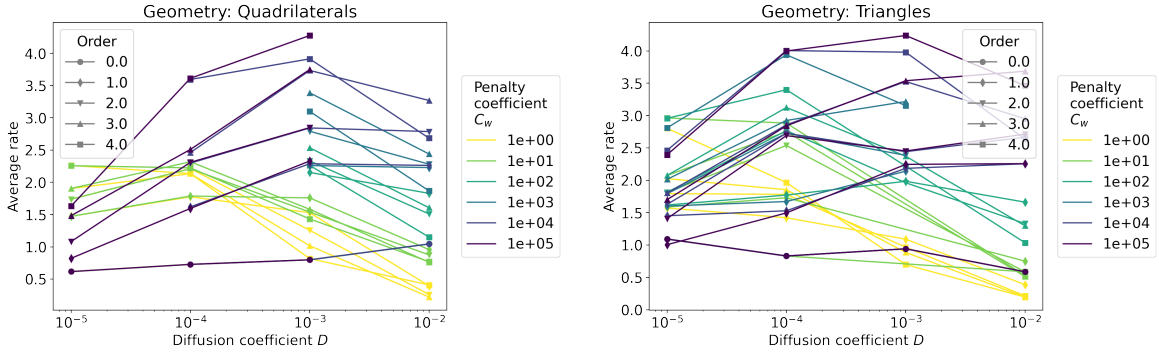


Figure 13: Example 5. Average convergence rate for different choices of C_w for quadrilaterals (left) and triangles (right).

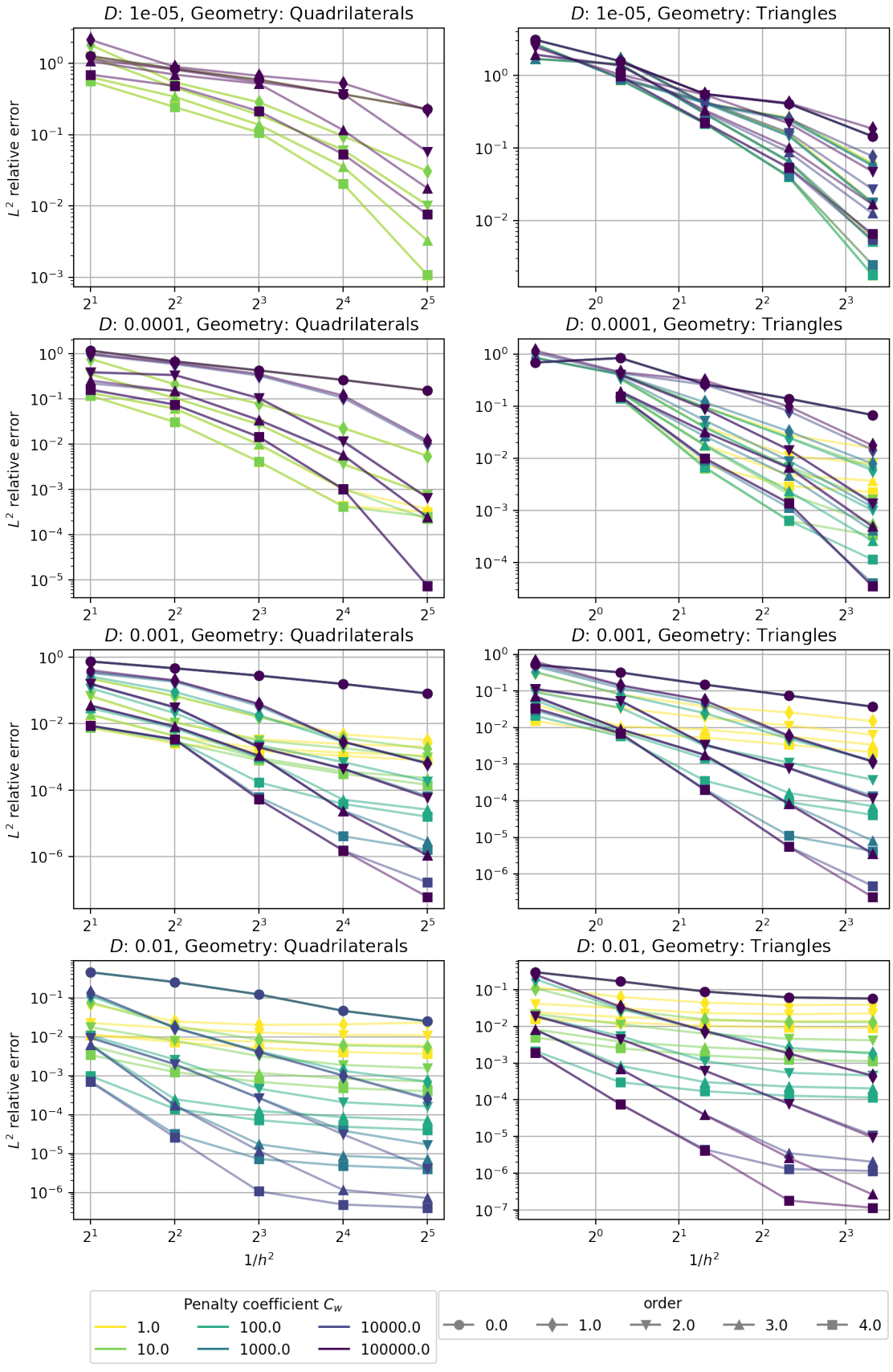
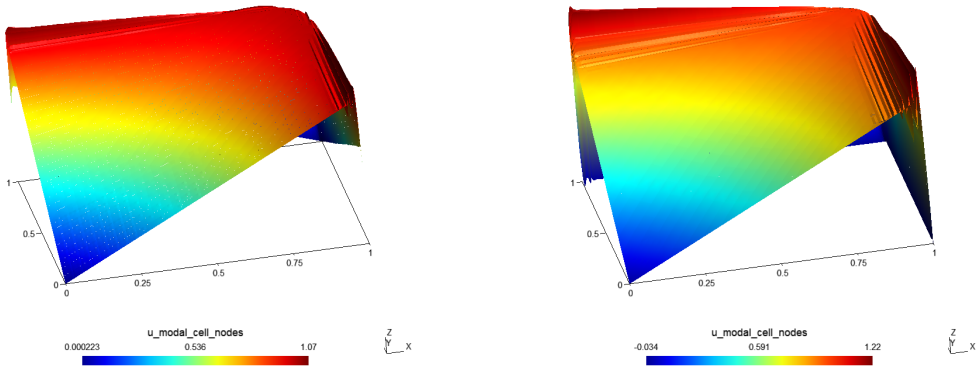


Figure 14: Example 5. Relative errors for different choices of C_w for quadrilaterals (left) and triangles (right).

Example 6 (Advection-diffusion 2D)**.** Based on Example 3 in [3], in $\Omega = [0,1]^2$ we will once again solve the equation (4.1.6) We set up the boundary condition and source function in such a way that the exact solution u_{exact} is

$$u_{exact} = -xy + x + y + \frac{\exp\left(-\frac{(x-1)(y-1)}{D}\right) - \exp\left(-\frac{1}{D}\right)}{\exp\left(-\frac{1}{D}\right) - 1}. \tag{4.2.13}$$

We omit analytical forms of g and boundary conditions for brevity, they can found in the code. Different values of the coefficient C_w in the penalty term yield different convergence behavior as demonstrated in Figures 16 and 17. Antonietti et al. [3] report convergence for order 1 and 2, these are in accord with ours.



(a) $C_w = 1$ (b) $C_w = 10^3$

Figure 15: Example 6. Solutions using quadrilateral mesh for $D = 0.001$ and for different values of C_w for quadrilaterals (left) and triangles (right).

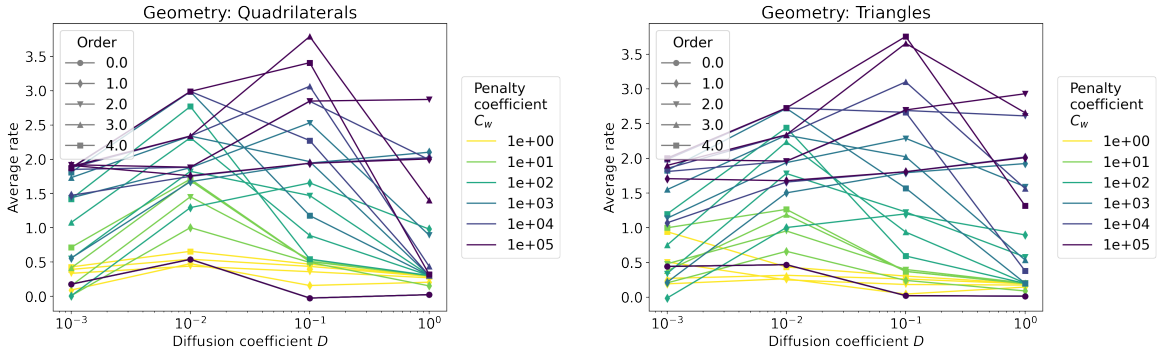


Figure 16: Example 6. Average convergence rate for different choices of C_w

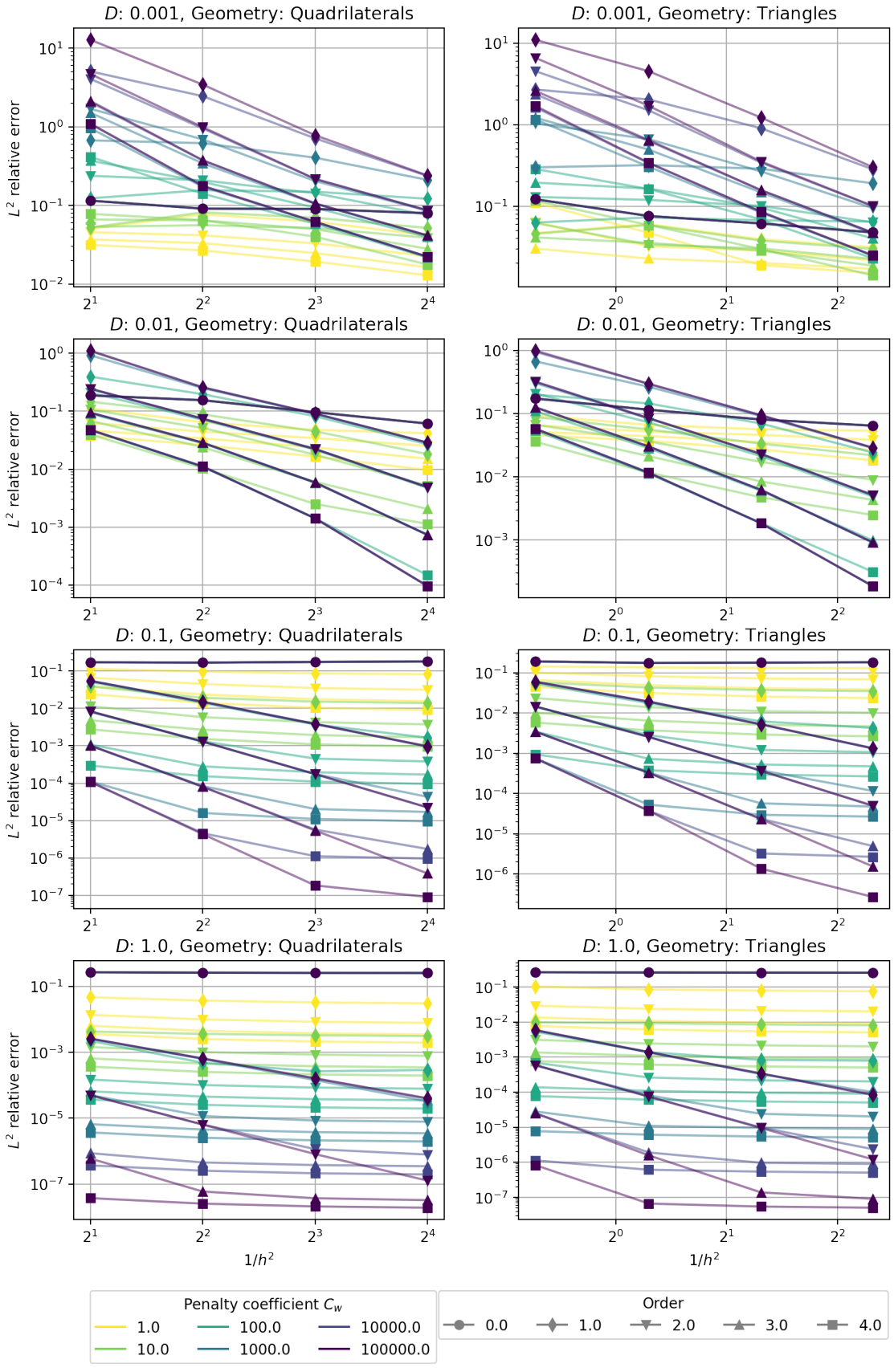


Figure 17: Example 6. Relative errors for different choice of C_w for quadrilaterals (left) and triangles (right).

Example 7 (Viscous Burgers 1D)**.** Based on [15, Section 7.1.2, Example 7.5, p. 255]. On $\Omega = [-1, 1]$ we will solve the viscous Burgers' equation (4.1.10) with the zero source function. This equation has an exact solution of a traveling wave

$$u_{exact} = -\tanh\left(-\frac{2\,t - 2\,x - 1}{4\,D}\right) + 1. \tag{4.2.14}$$

We set boundary conditions to match the solution

$$\begin{aligned} u(-1,t) &= -\tanh\left(-\frac{2\,t+1}{4\,D}\right) + 1, & u(-1,t) &= -\tanh\left(-\frac{2\,t-3}{4\,D}\right) + 1, \\ u_x(-1,t) &= \frac{1}{2D}\tanh\left(-\frac{2\,t+1}{4\,D}\right)^2 - \frac{1}{2D}, & u_x(1,t) &= \frac{1}{2D}\tanh\left(-\frac{2\,t-3}{4\,D}\right)^2 - \frac{1}{2D}. \end{aligned} \tag{4.2.15}$$

We will study the solution at time $t = 1$ with $D = 0.001$ and $D = 0.01$. Figure 20 shows relative errors for different combinations of parameters. In case $D = 0.001$ increasing C_w is detrimental to the accuracy of the solution as it develops into a steep step, see analytic solution in Figure 18, whose approximation requires discontinuity in the approximate solution. For $D = 0.01$ the diffusion

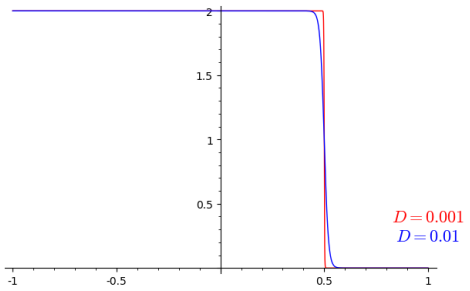
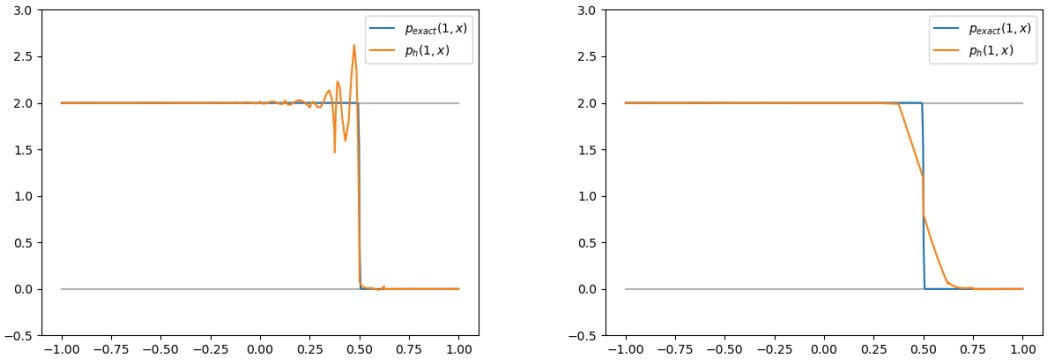


Figure 18: Example 7. Exact solution at $t = 1$.

leads to a smoother solution and the penalty term counteracting discontinuity between elements is beneficial, it also helps to counteract oscillations similarly to the limiter in Example 1. However, if the limiter is employed the smoothing of the solution caused by artificial diffusion is so severe that the penalty term is detrimental in either case. Figure 19 demonstrates the effect of the limiter.



(a) Limit: False

(b) Limit: True

Figure 19: Example 7. 4th order solution for $D = 0.001$, $C_w = 10$ with and without limiting, uniform mesh with 16 elements.

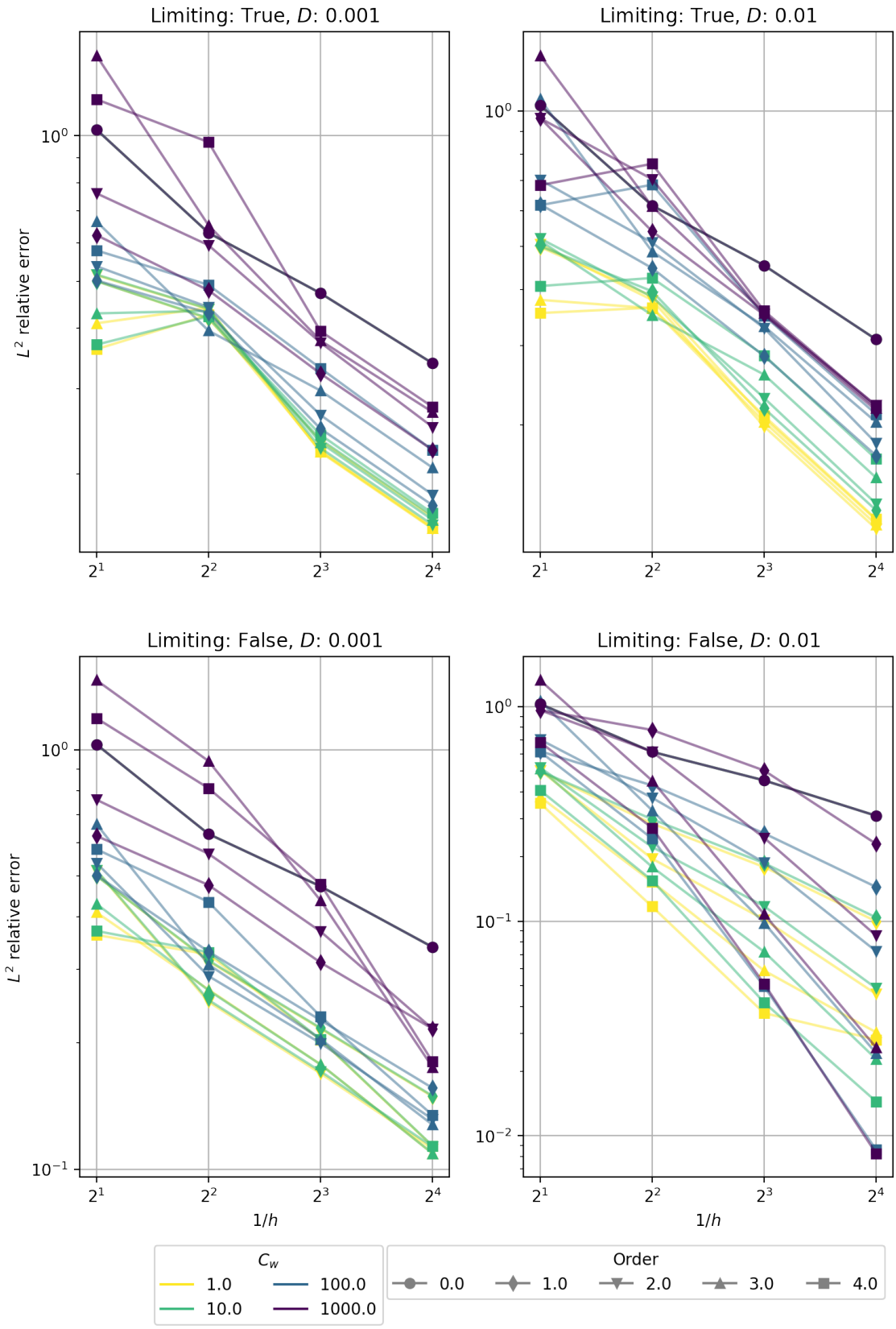
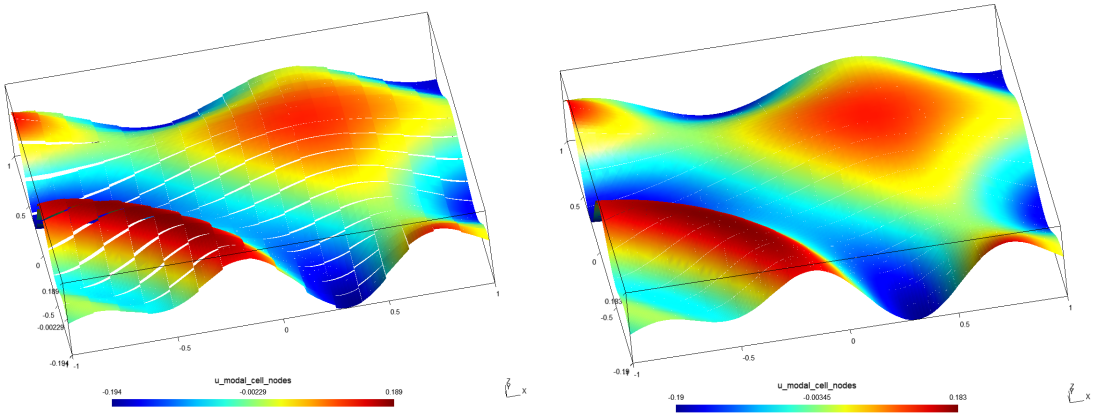


Figure 20: Example 7. Relative errors for different choices of C_w

Example 8 (Viscous Burgers 2D)**.** Based on example in [19, Section 1.6], we will solve the equation (4.1.10) in $\Omega = [0,1]^2$. We setup the boundary condition and source function in such way that the exact solution u_{exact} is

$$u_{exact} = -\left(e^{(-t)} - 1\right)(\sin(5\,xy) + \sin(-4\,xy + 4\,x + 4\,y)). \tag{4.2.16}$$

We omit analytical forms of g and boundary conditions for brevity. Different values of the coefficient C_w in the penalty term then yield only slightly different convergence behavior as demonstrated in Figure 22 and 23. In this case the solution does not feature any sharp steps and an increase in the penalty coefficient leads to an increase in accuracy. In Figure 21 this effect is clearly visible in the numerical solution, similarly to Example 4. Kučera in [19] reports average convergence rates for an irregular triangular mesh slightly higher then ours.



(a) $C_w = 1$ (b) $C_w = 15$

Figure 21: Example 8. Solution for different values of C_w on a quadrilateral mesh.

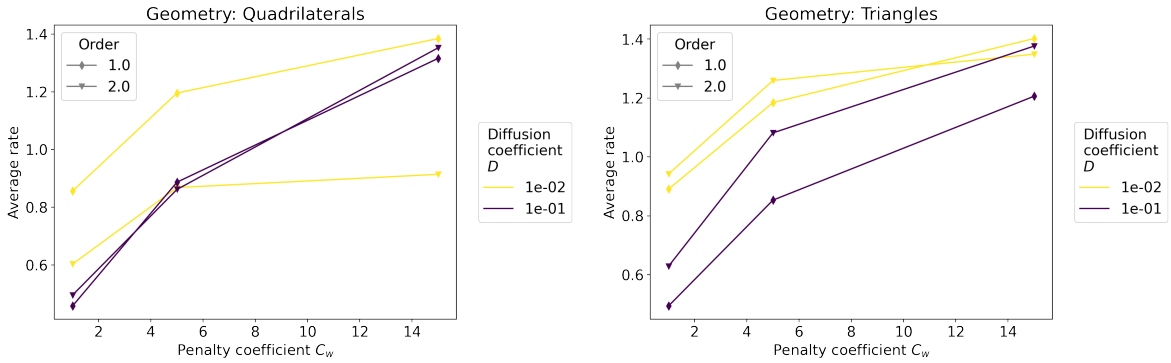


Figure 22: Example 8. Average convergence rates for different values of C_w for quadrilaterals (left) and triangles (right).

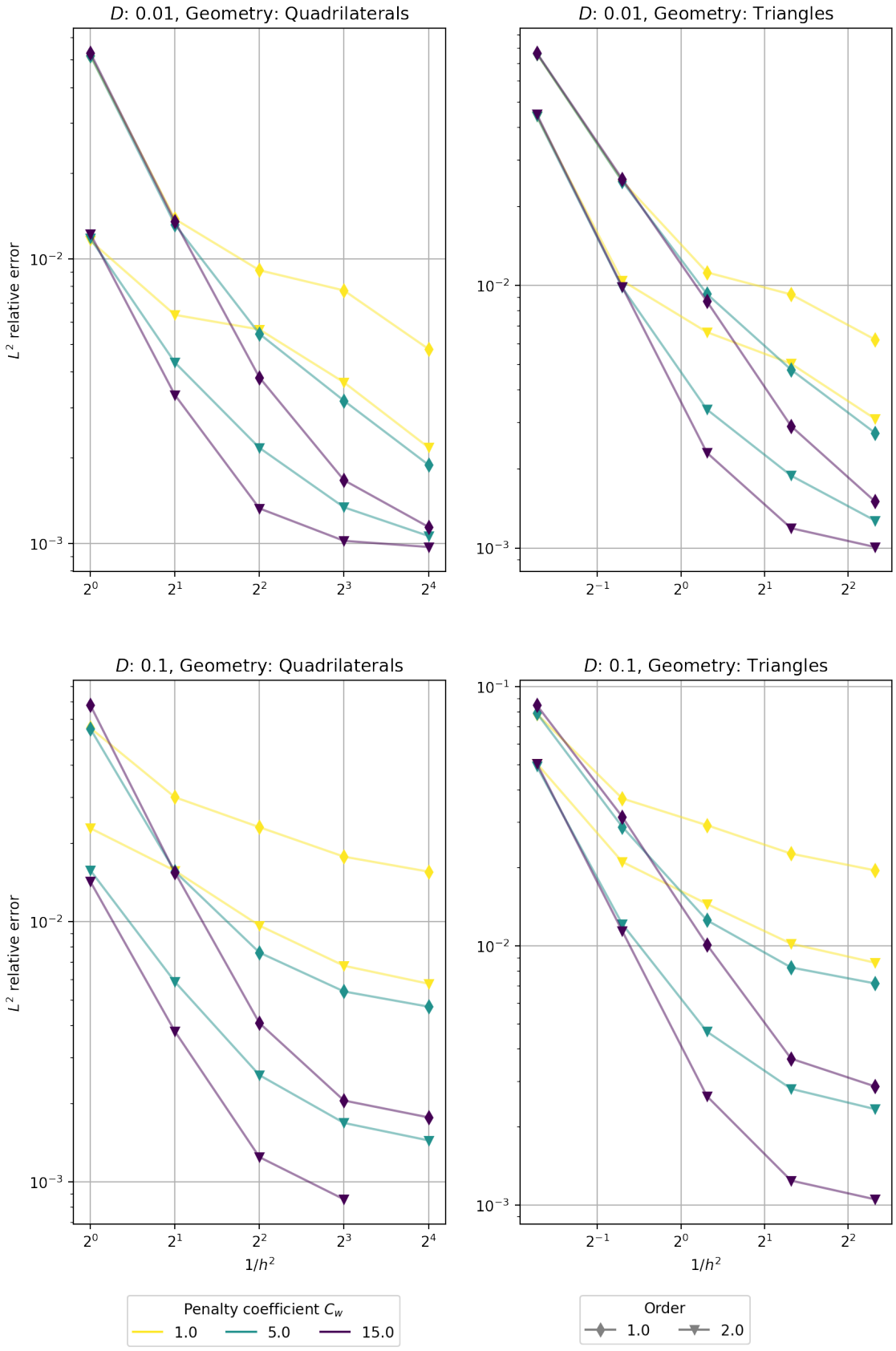


Figure 23: Example 8. Relative errors for different choices of C_w for quadrilaterals (left) and triangles (right).

Chapter 5

Conclusion

In this work, we implemented the discontinuous Galerkin method into *SfePy* package. *SfePy* uses a term based syntax for building discretization of equations. We implemented several new terms, namely the linear advection flux term, the general hyperbolic flux term, the diffusion flux and the diffusion penalty terms, and the general term for computing the integral

$$\int_{T^k} \vec{f}(P_i^k \psi_i) \cdot \nabla \psi_j.$$

Along with a wide range of terms already present in *SfePy* this allows users to discretize a variety of useful equations. To enable solving transient equations we implemented two explicit time-stepping solvers, the forward Euler solver and the TVD Runge-Kutta of the 3rd order solver. Moreover, we implemented the moment limiters for 1D and 2D transient problems. These contribution are part of the *SfePy* since the release 2020.2 .

To study properties of the method we calculated relative errors with respect to an analytical solution for seven example problems chosen from the literature. For some of them, we present results which complement already published parametric studies. In example 1 and 2 we demonstrated behavior of the moment limiter for 3rd and 4th order approximations. In examples 4, 5 and 6 we explored dependency between the optimal value of coefficient C_w in diffusion penalty term and diffusion coefficient for up to 4th order approximation. An in example Example 7 we expanded the analysis to include use of the limiter. Examples with the diffusion show the usefulness of the diffusion penalty term but also demonstrate its limits and provide direction when choosing value of coefficient C_w. The usage of the limiter introduces an artificial diffusion which significantly impacts quality of the solution. However, in some cases it is necessary to maintain the stability of the method. For transient problems, the performance of the method is further hindered by used time stepping solvers, nevertheless, the method still performs to the expectations.

This fulfills all the major goals of this work. There are, however, still many possible improvements and opportunities for future work. Although from the time and memory requirements perspective the implementation of the method scales well enough with *SfePy* capabilities, there is still room for improvement. Calls of `numpy.einsum` could use an optimized tensor contraction path, which could be retained between individual term evaluations (i.e., between time steps). The implementation of limiters is rather ad-hoc and refactoring it to bring it in line with the design of other *SfePy* elements would help to make their code more readable and their usage simpler and more versatile. One important feature available in *SfePy* missing from this DG FEM implementation is the ability to solve systems of PDEs. This lack could inspire future work as it would require substantial modification of flux terms as well as the implementation of strategies for evolving systems of interest like Euler or Navier-Stokes equations [15]. Further, besides the implemented Lax-Friedrichs flux, there is a variety of other numerical fluxes for example the Godunov flux [11] or fluxes designed specifically for solving Euler equations [19, Section 3.3]. These could be added along with terms for specifying Newton boundary conditions to broaden the selection

of tools available in *SfePy*. Thanks to the versatility of the problem specification this would also unlock a large potential for further study of the method behavior.

Acknowledgment

This work was partially supported by project GACR 16-03823S.

Bibliography

[1] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[2] P.R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Transactions on Mathematical Software*, 45:2:1–2:26, 2019.

[3] Paola Antonietti and Alfio Quarteroni. Numerical performance of discontinuous and stabilized continuous Galerkin methods for convection–diffusion problems. *Numerical Methods for Hyperbolic Equations*, pages 75–85, 2013.

[4] Onno Bokhove and Jaap JW van der Vegt. Introduction to (dis) continuous galerkin finite element methods. *Department of Applied Mathematics, University of Twente*, 2008.

[5] Christophe Chalons, Paola Goatin, and Luis M. Villada. High-Order Numerical Schemes for One-Dimensional Nonlocal Conservation Laws. *SIAM Journal on Scientific Computing*, 40(1):A288–A305, 2018.

[6] Robert Cimrman and SfePy developers. Main sfepy repository. Online 7. 5. 2020, 2020. https://github.com/sfepy/sfepy.

[7] Robert Cimrman and SfePy developers. SfePy: Simple Finite Elements in Python. Online 7. 5. 2020, 2020. http://sfepy.org.

[8] Robert Cimrman, Vladimír Lukeš, and Eduard Rohan. Multiscale finite element calculations in python using sfepy. *Advances in Computational Mathematics*, 45(4):1897–1921, 2019.

[9] Bernardo Cockburn and Chi-Wang Shu. Runge – Kutta Discontinuous Galerkin Methods for Convection-Dominated Problems. *Journal of Scientific Computing*, 16(3):173–261, 2001.

[10] The SciPy community. NumPy v1.18 Manual. Online 7. 5. 2020, 2020. https://numpy.org/doc/stable/reference/generated/numpy.einsum.html.

[11] Daniele Antonio Di Pietro and Alexandre Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*, volume 69 of *Mathématiques et Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[12] Vít Dolejší and Miloslav Feistauer. *Space-Time Discontinuous Galerkin Method*, pages 223–335. Springer International Publishing, Cham, 2015.

[13] Emmanuil H Georgoulis. Approximation Algorithms for Complex Systems. *Approximation Algorithms for Complex Systems*, 3:91 – 126, 2011.

[14] Sigal Gottlieb and Chi-Wang Shu. Total variation diminishing Runge-Kutta schemes. *Mathematics of Computation of the American Mathematical Society*, 67(221):73–85, 2002.

[15] Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods*, volume 54 of *Texts in Applied Mathematics*. Springer New York, New York, 2008.

[16] Gabriela Holubová and Pavel Drábek. *Parciální diferenciální rovnice*. University of West Bohemia, Faculty of Applied Science, 2011.

[17] Open Source Initiative. The 3-clause bsd license. Online 17.5.2020, 2020. https://opensource.org/licenses/BSD-3-Clause.

[18] Lilia Krivodonova. Limiters for high-order discontinuous Galerkin methods. *Journal of Computational Physics*, 226(1):879–896, 2007.

[19] Václav Kučera. *Higher order methods for the solution of compressible flows*. Doctoral thesis, Charles University in Prague, 2007.

[20] Lars Pesch, Alexander Bell, Henk Sollie, Vijaya R. Ambati, Onno Bokhove, and Jaap J. W. Van Der Vegt. Hpgem—a software framework for discontinuous galerkin finite element methods. *ACM Trans. Math. Softw.*, 33(4):23–es, August 2007.

[21] Jean-François Remacle, Nicolas Chevaugeon, Émilie Marchandise, and Christophe Geuzaine. Efficient visualization of high-order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4):750–771, 2007.

[22] Philip Roe. A Simple Explanation of Superconvergence for Discontinuous Galerkin Solutions to ut+ux=0. *Communications in Computational Physics*, 21(4):905–912, 2017.

[23] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[24] F.D. Witherden, A.M. Farrington, and P.E. Vincent. Pyfr: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028 – 3040, 2014.

[25] Ling Yuan and Chi-Wang Shu. Discontinuous galerkin method based on non-polynomial approximation spaces. *J. Comput. Phys.*, 218(1):295–323, October 2006.

[26] Tomáš Zítka. Example files and utilities for studying convergence of discontinous Galerkin method in SfePy. Online 17. 6. 2020, 2020. Zenodo. DOI: http://doi.org/10.5281/zenodo.3947773.