

FESTUNG: A MATLAB/GNU Octave toolbox for the discontinuous
Galerkin method. Part III: Hybridized discontinuous Galerkin (HDG) formulation
Non Peer-reviewed author version

JAUST, Alexander; Reuter, Balthasar; Aizinger, Vadym; SCHUETZ, Jochen &
Knabner, Peter (2018) FESTUNG: A MATLAB/GNU Octave toolbox for the
discontinuous Galerkin method. Part III: Hybridized discontinuous Galerkin (HDG)
formulation. In: COMPUTERS & MATHEMATICS WITH APPLICATIONS, 75 (12), p. 4505-4533.

DOI: 10.1016/j.camwa.2018.03.045

Handle: <http://hdl.handle.net/1942/25962>

FESTUNG: A MATLAB / GNU Octave toolbox for the discontinuous Galerkin method. Part III: Hybridized discontinuous Galerkin (HDG) formulation

Alexander Jaust^a, Balthasar Reuter^b, Vadym Aizinger^{c,b,*}, Jochen Schütz^a, Peter Knabner^b

^aUHasselt – Hasselt University, Faculty of Sciences, Agoralaan Gebouw D, 3590 Diepenbeek, Belgium

^bFriedrich–Alexander University of Erlangen–Nürnberg, Department of Mathematics, Cauerstraße 11, 91058 Erlangen, Germany

^cAlfred Wegener Institute, Helmholtz Centre for Polar and Marine Research, Am Handelshafen 12, 27570 Bremerhaven, Germany

Abstract

The third paper in our series on open source MATLAB / GNU Octave implementation of the discontinuous Galerkin (DG) method(s) focuses on a hybridized formulation. The main aim of this ongoing work is to develop rapid prototyping techniques covering a range of standard DG methodologies and suitable for small to medium sized applications. Our FESTUNG package relies on fully vectorized matrix / vector operations throughout, and all details of the implementation are fully documented. Once again, great care is taken to maintain a direct mapping between discretization terms and code routines as well as to ensure full compatibility to GNU Octave. The current work formulates a hybridized DG scheme for a linear advection problem, describes hybrid approximation spaces on the mesh skeleton, and compares the performance of this discretization to the standard (element-based) DG method for different polynomial orders.

Keywords: MATLAB, GNU Octave, hybridized discontinuous Galerkin (HDG) method, vectorization, open source, diagonally implicit Runge-Kutta method (DIRK)

1. Introduction

The discontinuous Galerkin (DG) method first introduced in the early 70s in [1] went on to have an illustrious career as one of the most popular numerical methods especially (but not exclusively) for fluid simulation and engendered a whole family of numerical schemes (see, e.g. [2, 3] and the references therein). The reasons for this success are many [4, 5]: an extremely flexible framework easily lending itself to many different types of equations, stability and conservation properties comparable to those of the finite volume method, natural support for high order discretizations and various types of adaptivity (h -, p -, r -) as well as for complex domain geometries, etc. Due to a favorable computation-to-communication ratio, the method also fits extremely well into the popular parallel and hybrid computational paradigms [6].

However, one aspect of the DG methodology places it at a clear disadvantage compared to the classical finite element and finite volume methods: a large number of degrees of freedom with corresponding memory requirements. This drawback becomes even more restrictive in time-implicit or stationary numerical solvers that rely on matrix assembly, where increases in the lengths of vectors of unknowns become quadratically compounded in the sizes of corresponding matrix blocks. One idea proposed to speed up the solution of linear systems resulting from DG discretizations exploits the specifics of DG approximation spaces. Since DG basis functions are usually element-local (and often also hierarchical), the linear system can be trivially split into parts corresponding to different polynomial

*Corresponding author

orders resulting in a scheme somewhat inspired by multigrid solvers, where different polynomial approximations on a fixed mesh play the role of fine and coarse mesh solutions of the classical multigrid. This approach can be carried out for each polynomial degree as in the p -multigrid method [7, 8, 9] or using a two scale technique as in the hierarchical scale separation (HSS) method or variations thereof [10, 11, 12, 13]. Another common way to deal with this issue—and in many cases even to speed up the time-to-solution [14, 15, 16]—is to use *hybridization*. Roughly speaking, this means that the discretized PDE is augmented with an unknown—let us call it λ_h —supported on the skeleton of the mesh consisting of element edges in 2D and element faces in 3D. Using static condensation on the algebraic system level produces a significantly smaller system than that obtained for an unhybridized DG method at the price of additional cell-wise (small and uncoupled) linear systems that have to be solved alongside the globally coupled system on the mesh skeleton. Since all local solves are element-local and fully decoupled, the parallel communication cost of this algorithm part is zero.

The idea of using hybridization can be traced back to the 60s [17], it has subsequently been used in the context of mixed methods [18, 19]. In those works, λ_h was not only considered an implementation feature but also as a way to obtain a more accurate solution via postprocessing. Based on the work in [20], Cockburn and coworkers introduced the hybridized discontinuous Galerkin method in a unifying framework in [21]. Subsequently, the method has been extended to various types of equations such as the Stokes [22, 23] and Darcy-Stokes equations [24], the incompressible and compressible Navier-Stokes equations [25, 26, 27, 28], the Maxwell equations [29], and—particularly relevant for this work—the convection(-diffusion) equation [30, 31, 32]. For an interesting unification framework, we would also like to draw reader’s attention to a recent publication [33].

The current work applies and extends to the hybridized schemes the framework and design principles introduced in [34, 35] for unhybridized DG formulations and epitomized in our MATLAB/GNU Octave toolbox *FESTUNG* (Finite Element Simulation Toolbox for UNstructured Grids) available at [36, 37]. Citing from [34], we aim to

1. Design a general-purpose software package using the DG method for a range of standard applications and provide this toolbox as a research and learning tool in the open source format.
2. Supply a well-documented, intuitive user-interface to ease adoption by a wider community of application and engineering professionals.
3. Relying on the vectorization capabilities of MATLAB/GNU Octave, optimize the computational performance of the toolbox components and demonstrate these software development strategies.
4. Maintain throughout full compatibility with GNU Octave to support users of open source software.

For details about basic data structures and a general overview of the solver structure, we refer the interested reader to our first publication [34], which applies the local discontinuous Galerkin (LDG) method to the diffusion operator. A DG discretization of the same model problem as in this work combined with higher-order explicit time stepping schemes and arbitrary order vertex-based slope limiters [38, 39] is presented in the second paper in series [35]. The implementation presented in the current publication makes use of a new solver structure tailored towards improved readability and maintainability of the code and designed to ease coupling of different solvers. A detailed description of this new structure with an outline of the coupling capabilities is in preparation [40].

The rest of the paper is structured as follows: The model problem is introduced in the next section accompanied by a detailed description of the space and time discretization in Section 3. Important aspects of the implementation including local mappings, numerical quadrature, and the assembly of the system matrix are presented in Section 4. In Section 5, the code is verified using analytical convergence tests, and the numerical results are compared to those of the unhybridized DG implementation of the model problem from our previous publication [35]. Section 6 lists the routines mentioned in this article, and Section 7 contains some conclusions and a brief outlook of future tasks.

2. Model problem

Let $J := (0, t_{\text{end}})$ be a finite time interval, and let $\Omega \subset \mathbb{R}^2$ be a polygonally bounded Lipschitz domain with boundary $\partial\Omega$. We solve the *linear advection equation*

$$\partial_t c(t, \mathbf{x}) + \nabla \cdot \mathbf{f}(t, \mathbf{x}, c(t, \mathbf{x})) = \xi(t, \mathbf{x}) \quad \text{in } J \times \Omega, \quad (1)$$

where the scalar quantity $c : J \times \Omega \rightarrow \mathbb{R}$ is unknown. The flux function $\mathbf{f} : J \times \Omega \times \mathbb{R} \rightarrow \mathbb{R}^2$ determines the type of the problem and may depend on time t and space coordinate \mathbf{x} . Within the context of this work, we assume

$$\mathbf{f}(t, \mathbf{x}, c(t, \mathbf{x})) := \begin{bmatrix} u^1(t, \mathbf{x}), & u^2(t, \mathbf{x}) \end{bmatrix}^T c(t, \mathbf{x}) \quad (2)$$

with given velocity $\mathbf{u} := [u^1, u^2]^T$ and source function $\xi : J \times \Omega \rightarrow \mathbb{R}$ independent of c . Additionally, Eq. (1) is equipped with some initial

$$c = c^0 \quad \text{on } \{0\} \times \Omega \quad (3)$$

and Dirichlet boundary condition

$$c = c_D \quad \text{on } J \times \partial\Omega_{\text{in}}(t) \quad (4)$$

on inflow boundaries $\partial\Omega_{\text{in}}(t) := \{\mathbf{x} \in \partial\Omega \mid \mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu}(\mathbf{x}) < 0\}$ with $\boldsymbol{\nu}(\mathbf{x})$ denoting the outward pointing unit normal vector. Since (1) specifies a first-order hyperbolic equation, no boundary conditions need to be prescribed on the outflow boundary $\partial\Omega_{\text{out}}(t) := \partial\Omega \setminus \partial\Omega_{\text{in}}(t)$. Functions $c^0 : \Omega \mapsto \mathbb{R}$ and $c_D : J \times \partial\Omega_{\text{in}}(t) \mapsto \mathbb{R}$ are given. The model problem is the same as the one considered in the second paper of our series [35]; however, a slightly different notation is used here.

3. Discretization

3.1. Notation

Let $\mathcal{T}_h = \{T\}$ be a regular family of non-overlapping partitions of a polygonally bounded domain Ω into K closed triangles T . The hybridized discontinuous Galerkin scheme employed in this work uses unknowns located on element boundaries, i.e., on the edges of elements (also referred to as the trace of the mesh). We introduce the set of all edges $\mathcal{E}_h = \mathcal{E}_{\text{int}} \cup \mathcal{E}_{\text{ext}}$ consisting of sets of interior edges \mathcal{E}_{int} and boundary edges \mathcal{E}_{ext} . The latter is split into inflow \mathcal{E}_{in} and outflow edges \mathcal{E}_{out} . Furthermore, on each edge $E_{\bar{k}} \in \mathcal{E}_h$, a unit normal $\boldsymbol{\nu}_{\bar{k}}$ is defined such that $\boldsymbol{\nu}_{\bar{k}}$ is exterior to element $T_{\bar{k}-}$, where $T_{\bar{k}-}$ is an arbitrarily chosen but fixed element adjoining $E_{\bar{k}}$. Obviously, boundary edges have only one adjacent element. For interior edges, the element opposite to $T_{\bar{k}-}$ is called $T_{\bar{k}+}$. **The minus sign in $T_{\bar{k}-}$ is suppressed when no confusion is possible.** The total number of edges is denoted by

$$\bar{K} := |\mathcal{E}_h|, \quad \text{and} \quad \Gamma = \bigcup_{\bar{k}=1}^{\bar{K}} E_{\bar{k}}$$

is the trace of the mesh. We refer to elements using **indices k , and** add bars $\bar{\cdot}$ (e.g., \bar{k} or \bar{K}) whenever we refer to edges to distinguish one from another. Additionally, we make use of the notation from the previous publications [34, 35] where we referred to edges of an element T_k as $E_{kn}, n \in \{1, 2, 3\}$. Note the difference between this element-local edge numbering (“ E_{kn} is the n th edge of the k th element”) and global edge numbering (“ $E_{\bar{k}}$ is the \bar{k} th edge in the mesh”).

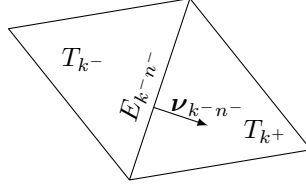


Figure 1: Two triangles adjacent to edge $E_{\bar{k}}$. It holds $E_{\bar{k}} = E_{k^- n^-} = E_{k^+ n^+}$ and $\nu_{k^- n^-} = -\nu_{k^+ n^+}$.

For the description of the method, we need mappings allowing us to switch back and forth between element-local and global indices. For that reason, we introduce a mapping $\rho(k, n)$ that relates the n th edge E_{kn} of element T_k to its global index in the set of edges \mathcal{E}_h

$$\rho : \{1, \dots, K\} \times \{1, 2, 3\} \rightarrow \{1, \dots, \bar{K}\}, \quad (k, n) \mapsto \bar{k}. \quad (5)$$

This mapping is not injective since for each interior edge there exists a pair of index tuples (k^-, n^-) and (k^+, n^+) that map to the same edge index \bar{k} , i. e., $E_{\bar{k}} = E_{k^- n^-} = E_{k^+ n^+}$ (cf. Fig. 1). We define a second mapping $\kappa(\bar{k}, l)$ to identify elements T_{k^-}, T_{k^+} adjacent to an edge $E_{\bar{k}}$. In this, argument $l \in \{1, 2\}$ denotes the edge-local index of the adjacent elements; it is constructed so that $l = 1$ refers to the “inner” element T_{k^-} (that always exists) while $l = 2$ refers to the “outer” element T_{k^+} that exists only for interior edges \mathcal{E}_{int} ,

$$\kappa : \{1, \dots, \bar{K}\} \times \{1, 2\} \rightarrow \{0, \dots, K\}, \quad (\bar{k}, l) \mapsto \begin{cases} k^- \in \{1, \dots, K\}, & \text{if } l = 1 \\ k^+ \in \{0, \dots, K\}, & \text{if } l = 2 \end{cases}. \quad (6)$$

As element indices start counting from 1, we set $\kappa(\bar{k}, 2) = 0, \forall E_{\bar{k}} \in \mathcal{E}_{\text{ext}}$ to mark the absence of the “outer” element.

3.2. Semi-discrete form and hybridization

The particular feature of the HDG method that distinguishes it from **conventional DG methods** is that solutions are not only sought on elements, but also on the skeleton of the mesh. To this end, one has to define approximation spaces on both \mathcal{T}_h and \mathcal{E}_h . Given $\mathbb{P}_p(T)$ and $\mathbb{P}_p(E)$, the spaces of complete polynomials of degree at most p on an element $T \in \mathcal{T}_h$ and edge $E \in \mathcal{E}_h$, respectively, we denote **global** broken polynomial spaces $\mathbb{P}_p(\mathcal{T}_h), \mathbb{P}_p(\mathcal{E}_h)$ on the triangulation \mathcal{T}_h and its set of edges \mathcal{E}_h by

$$\mathbb{P}_p(\mathcal{T}_h) := \{\varphi_h : \bar{\Omega} \rightarrow \mathbb{R} : \forall T \in \mathcal{T}_h, \varphi_h|_T \in \mathbb{P}_p(T)\}, \quad \mathbb{P}_p(\mathcal{E}_h) := \{\mu_h : \Gamma \rightarrow \mathbb{R} : \forall E \in \mathcal{E}_h, \mu_h|_E \in \mathbb{P}_p(E)\}. \quad (7)$$

To support the tutorial character of this paper, we briefly discuss the derivation of the HDG method for the advection equation and refer the reader to [31] for more details. The standard DG discretization of (1) is obtained by multiplying (1) with a test function $\varphi_h \in \mathbb{P}_p(\mathcal{T}_h)$, applying integration by parts on element $T_k \in \mathcal{T}_h$, and replacing the flux \mathbf{f} (see (2)) on element boundaries ∂T_k by a numerical flux function $\hat{\mathbf{f}}_{DG}(c_h^-, c_h^+)$ well-known from finite volume discretizations [41, 42, 43], where c_h^- denotes the interior and c_h^+ the exterior value of the discontinuous ansatz function c_h with respect to element $T_k \in \mathcal{T}_h$. Omitting the time argument of c_h in the flux function for the sake of readability, this yields

$$\int_{T_k} \partial_t c_h(t) \varphi_h \, d\mathbf{x} - \int_{T_k} \mathbf{f}(c_h) \cdot \nabla \varphi_h \, d\mathbf{x} + \int_{\partial T_k} \hat{\mathbf{f}}_{DG}(c_h^-, c_h^+) \cdot \nu_{T_k} \varphi_h^- \, ds = \int_{T_k} \xi(\mathbf{t}) \varphi_h \, d\mathbf{x} \quad \forall \varphi_h \in \mathbb{P}_p(\mathcal{T}_h). \quad (8)$$

At the boundary, c_h^+ has of course to be replaced by a suitably chosen boundary value $c_{\partial\Omega}(c_h^-)$ as in

$$c_{\partial\Omega}(c_h^-) = \begin{cases} c_h^-, & \text{on outflow boundary } \partial\Omega_{\text{out}}, \\ c_D, & \text{on inflow boundary } \partial\Omega_{\text{in}}. \end{cases} \quad (9)$$

In this work, the numerical flux is chosen to be a slightly modified local Lax-Friedrichs/Rusanov flux [44, 45]

$$\hat{\mathbf{f}}_{DG}(c_h^-, c_h^+) := \mathbf{f}\left(\frac{c_h^- + c_h^+}{2}\right) + \frac{\alpha}{2}(c_h^- - c_h^+) \mathbf{v}_{T_k}; \quad (10)$$

however, other choices are possible [33]. The stabilization parameter α must satisfy $\alpha \geq \{\max\{|\mathbf{f}(c_h^-) \cdot \mathbf{v}_{T_k}|, |\mathbf{f}(c_h^+) \cdot \mathbf{v}_{T_k}|\}\}$. In this work α is a user-defined constant that is the same for every edge. At the boundary, we simply replace the flux by $\mathbf{f}(c_{\partial\Omega}(c_h^-))$, see also (9). One can rewrite the numerical flux (10) to formally only depend on the mean value of c_h^- and c_h^+ and, e.g., on c_h^- : Upon defining a quantity $\lambda_h(t)$ on an edge $E_{\bar{k}}$ as

$$\lambda_h(t) := \begin{cases} \frac{1}{2}(c_h^-(t) + c_h^+(t)), & E_{\bar{k}} \in \mathcal{E}_{\text{int}}, \\ c_{\partial\Omega}(c_h^-(t)), & E_{\bar{k}} \in \mathcal{E}_{\text{ext}}, \end{cases} \quad (11)$$

the flux $\hat{\mathbf{f}}_{DG}$ in (10) can also be written as

$$\hat{\mathbf{f}}(\lambda_h, c_h^-) := \begin{cases} \mathbf{f}(\lambda_h) - \alpha(\lambda_h - c_h^-) \mathbf{v}_{T_k}, & E_{\bar{k}} \in \mathcal{E}_{\text{int}}, \\ \mathbf{f}(\lambda_h), & E_{\bar{k}} \in \mathcal{E}_{\text{ext}}. \end{cases} \quad (12)$$

Note that λ_h on an interior edge is a polynomial of degree p . Discretizing the strong formulation of λ_h given in (11) in a DG framework is straightforward. Together with the slightly modified (only the flux function differs!) version of (8) this yields the HDG discretization: Seek $(c_h(t), \lambda_h(t)) \in \mathbb{P}_p(\mathcal{T}_h) \times \mathbb{P}_p(\mathcal{E}_h)$ such that for all $t \in J$ there holds

$$\int_{T_k} \partial_t c_h(t) \varphi_h \, d\mathbf{x} - \int_{T_k} \mathbf{f}(c_h) \cdot \nabla \varphi_h \, d\mathbf{x} + \int_{\partial T_k} \hat{\mathbf{f}}(\lambda_h, c_h^-) \cdot \mathbf{v}_{T_k} \varphi_h^- \, ds = \int_{T_k} \xi \varphi_h \, d\mathbf{x}, \quad \forall \varphi_h \in \mathbb{P}_p(\mathcal{T}_h), \quad (13a)$$

$$\int_{E_{\bar{k}} \in \mathcal{E}_{\text{int}}} \alpha \mu_h (2\lambda_h(t) - c_h^-(t) - c_h^+(t)) \, ds + \int_{E_{\bar{k}} \in \mathcal{E}_{\text{ext}}} \mu_h (\lambda_h(t) - c_{\partial\Omega}(c_h^-(t))) \, ds = 0, \quad \forall \mu_h \in \mathbb{P}_p(\mathcal{E}_h). \quad (13b)$$

Note that α in (13b) has been introduced to make the formulation compatible to the one used for viscous problems [31].

It can immediately be seen that the above formulation is nothing else than our point-of-departure DG method, see (8), written in another, yet equivalent, way. At first glance, this scheme does not seem to have any advantages compared to other DG schemes: on the contrary, an additional equation and an additional unknown λ_h are apparent. However, the choice of the numerical flux on element boundaries leads to an inter-element coupling solely by the function λ_h . In Sec. 3.4, we explain in detail how this structure of the system can be exploited to reduce the number of globally coupled unknowns, which turns out to be especially attractive for discretizations that rely on implicit solution techniques such as time-implicit schemes or stationary problems. To give a first glimpse of this procedure consider the stationary part of (13) (i.e. $\partial_t c_h = 0$). This results in a linear system of equations of form

$$\begin{pmatrix} \bar{\mathbf{L}} & \bar{\mathbf{M}} \\ \mathbf{N} & \mathbf{P} \end{pmatrix} \begin{pmatrix} \mathbf{C} \\ \boldsymbol{\Lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{B}_\varphi \\ \mathbf{B}_\mu \end{pmatrix}$$

with matrices $\bar{\mathbf{L}} \in \mathbb{R}^{d_{\mathcal{T}_h} \times d_{\mathcal{T}_h}}$, $\bar{\mathbf{M}} \in \mathbb{R}^{d_{\mathcal{T}_h} \times d_{\mathcal{E}_h}}$, $\mathbf{N} \in \mathbb{R}^{d_{\mathcal{E}_h} \times d_{\mathcal{T}_h}}$ and $\mathbf{P} \in \mathbb{R}^{d_{\mathcal{E}_h} \times d_{\mathcal{E}_h}}$, where $d_{\mathcal{T}_h} = \dim(\mathbb{P}_h(\mathcal{T}_h))$ and $d_{\mathcal{E}_h} = \dim(\mathbb{P}_h(\mathcal{E}_h))$. \mathbf{C} and $\boldsymbol{\Lambda}$ are basis coefficients corresponding to c_h and λ_h , respectively, see (15). It is important to note that due to the inter-element coupling in λ_h only, the matrix $\bar{\mathbf{L}}$ (possibly reordered) is block-diagonal and can therefore be easily inverted. Thus, \mathbf{C} can be computed by

$$\mathbf{C} = \bar{\mathbf{L}}^{-1} (\mathbf{B}_\varphi - \bar{\mathbf{M}} \boldsymbol{\Lambda}),$$

which can be plugged into the second equation to obtain

$$\bar{\mathbf{N}} \bar{\mathbf{L}}^{-1} (\mathbf{B}_\varphi - \bar{\mathbf{M}} \boldsymbol{\Lambda}) + \mathbf{P} \boldsymbol{\Lambda} = \mathbf{B}_\mu.$$

This is an algebraic system in $\mathbf{\Lambda}$ only, that for higher order polynomial spaces tends to be much smaller than the corresponding linear system for an unhybridized DG scheme of the same order. This has a tremendous influence on the efficiency of iterative solvers and is one of the reasons for the success of the HDG method.

In the following sections, we make these introductory remarks more specific and explain their implementation in the software.

3.2.1. Local basis representation

The DG function spaces defined in (7) do not have any continuity constraints across element or edge boundaries. Consequently, a two-dimensional basis function $\varphi_{ki} : \bar{\Omega} \rightarrow \mathbb{R}$ for $\mathbb{P}_p(\mathcal{T}_h)$ is only supported on $T_k \in \mathcal{T}_h$ and must fulfill

$$\forall k \in \{1, \dots, K\}, \quad \mathbb{P}_p(T_k) = \text{span}\{\varphi_{ki}\}_{i \in \{1, \dots, N\}} \quad \text{with} \quad N := \frac{(p+1)(p+2)}{2}.$$

In the same way, a one-dimensional basis function $\mu_{\bar{k}i} : \Gamma \rightarrow \mathbb{R}$ for $\mathbb{P}_p(\mathcal{E}_h)$ is only supported on the edge $E_{\bar{k}} \subset \Gamma$, ensuring

$$\forall \bar{k} \in \{1, \dots, \bar{K}\}, \quad \mathbb{P}_p(E_{\bar{k}}) = \text{span}\{\mu_{\bar{k}i}\}_{i \in \{1, \dots, \bar{N}\}} \quad \text{with} \quad \bar{N} := p+1.$$

We denote the numbers of local degrees of freedom on a triangle or an edge by N or \bar{N} , respectively. In this work, we choose orthonormal, hierarchical Legendre polynomials as basis functions. For details and closed-form expressions of the two-dimensional basis functions φ_{ki} we refer to our first paper [34] (those can be easily obtained from an arbitrary polynomial basis via e.g. Gram-Schmidt orthogonalization procedure using a symbolic algebra software). The one-dimensional basis functions up to order four given on the unit interval $[0, 1]$ are defined as

$$\begin{aligned} \hat{\mu}_1(s) &:= 1, & \hat{\mu}_2(s) &:= \sqrt{3}(1-2s), & \hat{\mu}_3(s) &:= \sqrt{5}(1-6s+6s^2), \\ \hat{\mu}_4(s) &:= \sqrt{7}(-1+12s-30s^2+20s^3), & \hat{\mu}_5(s) &:= \sqrt{9}(1-20s+90s^2-140s^3+70s^4). \end{aligned} \quad (14)$$

The local discrete solutions c_h on $T_k \in \mathcal{T}_h$ and λ_h on $E_{\bar{k}} \in \mathcal{E}_h$ are represented using local bases on elements and edges

$$c_h(t, \mathbf{x})|_{T_k} = \sum_{j=1}^N C_{kj}(t) \varphi_{kj}(\mathbf{x}), \quad \lambda_h(t, \mathbf{x})|_{E_{\bar{k}}} = \sum_{j=1}^{\bar{N}} \Lambda_{\bar{k}j}(t) \mu_{\bar{k}j}(\mathbf{x}). \quad (15)$$

3.2.2. System of equations

Testing (13a) with $\varphi_h = \varphi_{ki}$, $i = 1, \dots, N$ yields a time-dependent system of equations with the contribution from T_k given by

$$\begin{aligned} & \underbrace{\partial_t \sum_{j=1}^N C_{kj}(t) \int_{T_k} \varphi_{kj} \varphi_{ki} \, d\mathbf{x}}_{I \, (\mathbf{M}_\varphi)} - \underbrace{\sum_{j=1}^N C_{kj}(t) \sum_{m=1}^2 \int_{T_k} u^m(t, \mathbf{x}) \varphi_{kj} \partial_{x^m} \varphi_{ki} \, d\mathbf{x}}_{II \, (-\mathbf{G}^1 \text{ and } -\mathbf{G}^2)} + \underbrace{\sum_{j=1}^{\bar{N}} \Lambda_{\bar{k}j}(t) \int_{\partial T_k \setminus \partial \Omega} (\mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu}) \mu_{\bar{k}j} \varphi_{ki} \, ds}_{III \, (\mathbf{S})} \\ & - \underbrace{\alpha \sum_{j=1}^{\bar{N}} \Lambda_{\bar{k}j}(t) \int_{\partial T_k \setminus \partial \Omega} \mu_{\bar{k}j} \varphi_{ki} \, ds}_{IV \, (\alpha \mathbf{R}_\mu)} + \underbrace{\alpha \sum_{j=1}^N C_{kj}(t) \int_{\partial T_k \setminus \partial \Omega} \varphi_{kj} \varphi_{ki} \, ds}_{V \, (\alpha \mathbf{R}_\varphi)} \\ & + \underbrace{\int_{\partial T_k \cap \partial \Omega} \varphi_{ki} (\mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu}) \left\{ \begin{array}{ll} \sum_{j=1}^{\bar{N}} \Lambda_{\bar{k}j}(t) \mu_{\bar{k}j} & \text{on } \partial \Omega_{\text{out}} \\ c_D(t, \mathbf{x}) & \text{on } \partial \Omega_{\text{in}} \end{array} \right\} ds}_{VI \, (\mathbf{S}_{\text{out}} \text{ and } \mathbf{F}_{\varphi, \text{in}})} = \underbrace{\int_{T_k} \xi(t) \varphi_{ki} \, d\mathbf{x}}_{VII \, (\mathbf{H})}. \end{aligned} \quad (16a)$$

In (2) \mathbf{f} was defined as a linear function. Upon testing with $\mu_h = \mu_{\bar{k}i}$, the semi-discrete form of (13b) is then given by

$$\begin{aligned}
& \underbrace{\alpha \sum_{j=1}^{\bar{N}} \Lambda_{kj}(t) \int_{\partial T_k \setminus \partial \Omega} \mu_{\bar{k}j} \mu_{\bar{k}i} \, ds}_{VIII \ (\alpha \bar{\mathbf{M}}_\mu)} - \underbrace{\alpha \sum_{j=1}^N C_{kj}(t) \int_{\partial T_k \setminus \partial \Omega} \varphi_{kj} \mu_{\bar{k}i} \, ds}_{IX \ (-\alpha \mathbf{T})} \\
& + \underbrace{\sum_{j=1}^{\bar{N}} \Lambda_{kj}(t) \int_{\partial T_k \cap \partial \Omega} \mu_{\bar{k}j} \mu_{\bar{k}i} \, ds}_{X \ (\bar{\mathbf{M}}_\mu)} - \underbrace{\int_{\partial T_k \cap \partial \Omega} \mu_{\bar{k}i} \left\{ \begin{array}{ll} \sum_{j=1}^N C_{kj}(t) \varphi_{kj} & \text{on } \partial \Omega_{\text{out}} \\ c_D(t, \mathbf{x}) & \text{on } \partial \Omega_{\text{in}} \end{array} \right\} ds}_{XI \ (-\mathbf{K}_{\mu, \text{out}} \text{ and } \mathbf{K}_{\mu, \text{in}})} = 0
\end{aligned} \tag{16b}$$

We can rewrite system (16) in matrix form

$$\mathbf{M}_\varphi \partial_t \mathbf{C}(t) + \underbrace{(-\mathbf{G}^1(t) - \mathbf{G}^2(t) + \alpha \mathbf{R}_\varphi)}_{=\bar{\mathbf{L}}(t)} \mathbf{C}(t) + \underbrace{(\mathbf{S}(t) + \mathbf{S}_{\text{out}}(t) - \alpha \mathbf{R}_\mu)}_{=\bar{\mathbf{M}}(t)} \boldsymbol{\Lambda}(t) = \underbrace{\mathbf{H}(t) - \mathbf{F}_{\varphi, \text{in}}(t)}_{=\mathbf{B}_\varphi(t)}, \tag{17a}$$

$$\underbrace{(-\alpha \mathbf{T} - \mathbf{K}_{\mu, \text{out}}(t))}_{=\mathbf{N}(t)} \mathbf{C}(t) + \underbrace{(\alpha \bar{\mathbf{M}}_\mu + \tilde{\mathbf{M}}_\mu)}_{=\mathbf{P}} \boldsymbol{\Lambda}(t) = \underbrace{-\mathbf{K}_{\mu, \text{in}}(t)}_{=\mathbf{B}_\mu(t)} \tag{17b}$$

with representation vectors

$$\mathbf{C}(t) := [C_{11}(t) \cdots C_{1N}(t) \cdots \cdots C_{K1}(t) \cdots C_{KN}(t)]^T, \quad \boldsymbol{\Lambda} := [\Lambda_{11}(t) \cdots \Lambda_{1\bar{N}}(t) \cdots \cdots \Lambda_{\bar{K}1}(t) \cdots \Lambda_{\bar{K}\bar{N}}(t)]^T.$$

Recall that the inter-element coupling of the solution c_h in system (17) acts only via the edge function λ_h via $\bar{\mathbf{M}}(t) \in \mathbb{R}^{KN \times \bar{K}\bar{N}}$ and $\mathbf{N} \in \mathbb{R}^{\bar{K}\bar{N} \times KN}$ – block matrices generally combining rectangular blocks into rectangular block structures. The rectangular $N \times \bar{N}$ (or transposed) block shapes arise from the difference in the number of local degrees of freedom on elements (N) and edges (\bar{N}) for $p > 0$ (i. e., $N > \bar{N}$), whereas the rectangular block-matrix structure is the result of the difference between the number of edges (\bar{K}) and the number of elements (K) in the triangulation. Thus, $K \times \bar{K}$ such rectangular $N \times \bar{N}$ blocks build up the global matrix, resulting in a global $KN \times \bar{K}\bar{N}$ matrix $\bar{\mathbf{M}}$ (or \mathbf{N} with all dimensions transposed).

3.2.3. Contributions from volume terms I, II and VII

All matrices in system (17) have sparse block structure, and we define the non-zero entries in the remainder of this section. For all remaining entries we tacitly assume zero fill-in.

The integral of term I gives the standard mass matrix $\mathbf{M}_\varphi \in \mathbb{R}^{KN \times KN}$ with components defined as

$$[\mathbf{M}_\varphi]_{(k-1)N+i, (k-1)N+j} := \int_{T_k} \varphi_{kj} \varphi_{ki} \, d\mathbf{x}. \tag{18a}$$

This leads to a block diagonal matrix because basis and test functions φ_{ki} , $i \in \{1, \dots, N\}$ are only supported on element T_k . Therefore

$$\mathbf{M}_\varphi = \begin{bmatrix} \mathbf{M}_{\varphi, T_1} & & \\ & \ddots & \\ & & \mathbf{M}_{\varphi, T_N} \end{bmatrix} \quad \text{with local mass matrix} \quad \mathbf{M}_{\varphi, T_k} := \int_{T_k} \begin{bmatrix} \varphi_{k1} \varphi_{k1} & \cdots & \varphi_{k1} \varphi_{kN} \\ \vdots & \ddots & \vdots \\ \varphi_{kN} \varphi_{k1} & \cdots & \varphi_{kN} \varphi_{kN} \end{bmatrix} d\mathbf{x} \in \mathbb{R}^{N \times N}. \tag{18b}$$

We abbreviate $\mathbf{M}_\varphi = \text{diag}(\mathbf{M}_{\varphi, T_1}, \dots, \mathbf{M}_{\varphi, T_N})$.

The definition of block matrices $\mathbf{G}^m \in \mathbb{R}^{KN \times KN}$, $m \in \{1, 2\}$ differs slightly from the form in our previous publication [35] because we do not use the projected advection velocity but rather evaluate the velocity function/flux at each quadrature point. The component-wise entries are given by

$$[\mathbf{G}^m]_{(k-1)N+i, (k-1)N+j} = \int_{T_k} u^m \varphi_{kj} \partial_{x^m} \varphi_{ki} \, d\mathbf{x} \quad (19a)$$

again leading to a block-diagonal matrix $\mathbf{G}^m = \text{diag}(\mathbf{G}_{T_1}^m, \dots, \mathbf{G}_{T_K}^m)$, where each block is given by

$$\mathbf{G}_{T_K}^m := \int_{T_K} u^m \begin{bmatrix} \varphi_{k1} \partial_{x^m} \varphi_{k1} & \dots & \varphi_{kN} \partial_{x^m} \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1} \partial_{x^m} \varphi_{kN} & \dots & \varphi_{kN} \partial_{x^m} \varphi_{kN} \end{bmatrix} d\mathbf{x}. \quad (19b)$$

The source term enters the discretization as an additional term $\mathbf{H} \in \mathbb{R}^{KN}$ on the right-hand-side of the equation. The entries are given as

$$[\mathbf{H}]_{(k-1)N+i} = \int_{T_k} \xi \varphi_{ki} \, d\mathbf{x} \quad (20a)$$

such that the full vector is easily assembled as

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{T_1} \\ \vdots \\ \mathbf{H}_{T_K} \end{bmatrix} \quad \text{with} \quad \mathbf{H}_{T_k} = \int_{T_k} \xi \begin{bmatrix} \varphi_{k1} \\ \vdots \\ \varphi_{kN} \end{bmatrix} d\mathbf{x}. \quad (20b)$$

3.2.4. Contribution of edge terms III, IV, V, VI—first equation

Compared to the DG discretization used in our previous works [34, 35], the number of edge integrals has significantly increased. This is caused by the following factors:

1. Edge integrals are split into integrals over interior edges and over edges on the domain boundaries.
2. The numerical flux function (12) contains three terms compared to only one for the upwind flux used in [35].
3. An additional unknown is introduced that is only defined on edges resulting in an additional equation (13b).

To improve readability, we split the presentation of edge integrals into two sections: first, edge terms in the original equation for c_h and then the edge terms in the hybrid equation. Throughout the assembly description and within the implementation we use the *element-based view*, i.e., all edge terms are presented in a form allowing for the assembly as nested loops over elements T_k , $k \in \{1, \dots, K\}$ and then edges E_{kn} , $n \in \{1, 2, 3\}$ of each element. This is different from the *edge based view* which would allow to assemble the terms in a single loop over all edges $E_{\bar{k}}$, $\bar{k} \in \{1, \dots, \bar{K}\}$. We made this choice since the data structures in FESTUNG favor the element-based view. For that reason, from now on, we always consider $\mu_{knj} = \mu_{\bar{k}j}$ using the mapping $\rho : (k, n) \mapsto \bar{k}$ specified in (5).

Term III contributes to matrix $\mathbf{S} \in \mathbb{R}^{KN \times \bar{K}\bar{N}}$ as

$$[\mathbf{S}]_{(k-1)N+i, (\bar{k}-1)\bar{N}+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} (\mathbf{u} \cdot \mathbf{v}_{kn}) \mu_{knj} \varphi_{ki} \, ds. \quad (21a)$$

The entries are structured into $N \times \bar{N}$ -blocks contributed by each edge on every element given as

$$\mathbf{S}_{E_{kn}} = \int_{E_{kn}} (\mathbf{u} \cdot \mathbf{v}_{kn}) \begin{bmatrix} \mu_{kn1} \varphi_{k1} & \dots & \mu_{kn\bar{N}} \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \mu_{kn1} \varphi_{kN} & \dots & \mu_{kn\bar{N}} \varphi_{kN} \end{bmatrix} ds. \quad (21b)$$

Note that this is the contribution of a single interior edge $E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}$ of triangle T_k . As this integral is also evaluated on the neighboring element, this block will produce *two* contributions to matrix \mathbf{S} .

The matrix from term IV is a block matrix $\mathbf{R}_\mu \in \mathbb{R}^{KN \times \bar{K}\bar{N}}$ with blocks of size $N \times \bar{N}$ given by

$$[\mathbf{R}_\mu]_{(k-1)N+i, (\bar{k}-1)\bar{N}+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} \mu_{knj} \varphi_{ki} \, ds \quad (22a)$$

with $i \in \{1, \dots, N\}$ and $j \in \{1, \dots, \bar{N}\}$. The local matrix of a single edge reads

$$\mathbf{R}_{\mu, E_{kn}} = \int_{E_{kn}} \begin{bmatrix} \mu_{kn1} \varphi_{k1} & \dots & \mu_{kn\bar{N}} \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \mu_{kn1} \varphi_{kN} & \dots & \mu_{kn\bar{N}} \varphi_{kN} \end{bmatrix} ds. \quad (22b)$$

Term V gives another block diagonal contribution $\mathbf{R}_\varphi \in \mathbb{R}^{KN \times KN}$, where each entry is given by

$$[\mathbf{R}_\varphi]_{(k-1)N+i, (k-1)N+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} \varphi_{kj} \varphi_{ki} \, ds. \quad (23a)$$

The element-local matrix is then

$$\mathbf{R}_{\varphi, T_k} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} \begin{bmatrix} \varphi_{k1} \varphi_{k1} & \dots & \varphi_{kN} \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1} \varphi_{kN} & \dots & \varphi_{kN} \varphi_{kN} \end{bmatrix} ds, \quad (23b)$$

where each edge E_{kn} of each triangle T_k is visited exactly once, and $\mathbf{R}_\varphi = \text{diag}(\mathbf{R}_{\varphi, T_1}, \dots, \mathbf{R}_{\varphi, T_K})$. This is slightly different from the previous work [35], where the test functions φ^- and φ^+ from two neighboring elements may have been multiplied because elements would be coupled directly with each other.

Term VI incorporates the boundary conditions on boundary edges $E_{\bar{k}} \in \mathcal{E}_{\text{ext}}$. In the case of an inflow boundary condition, this contributes to the right-hand-side. Each entry of vector $\mathbf{F}_{\varphi, \text{in}} \in \mathbb{R}^{KN}$ is given as

$$[\mathbf{F}_{\varphi, \text{in}}]_{(k-1)N+i} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{in}}} \int_{E_{kn}} (\mathbf{u} \cdot \mathbf{v}_{kn}) c_D \varphi_{ki} \, ds. \quad (24)$$

Outflow boundary conditions depend on λ_h and therefore on the solution. This gives us an additional contribution $\mathbf{S}_{\text{out}} \in \mathbb{R}^{KN \times \bar{K}\bar{N}}$ to the left hand side, where each entry is given by

$$[\mathbf{S}_{\text{out}}]_{(k-1)N+i, (\bar{k}-1)\bar{N}+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{out}}} \int_{E_{kn}} (\mathbf{u} \cdot \mathbf{v}_{kn}) \mu_{knj} \varphi_{ki} \, ds. \quad (25)$$

This is almost identical to (21a) with the only difference being the set of edges considered, and thus the sub-blocks take the same form as in equation (21b). In the implementation, \mathbf{S} and \mathbf{S}_{out} are assembled together.

3.2.5. Contribution of edge terms VIII, IX, X, and XI — hybrid equation

The first term — term VIII — is very similar to an edge mass matrix with the only differences being the parameter α that has to be respected and the fact that every edge is visited twice because it is an integral over interior edges $E_{\bar{k}} \in \mathcal{E}_{\text{int}}$. $\bar{\mathbf{M}}_\mu \in \mathbb{R}^{\bar{K}\bar{N} \times \bar{K}\bar{N}}$ is given by

$$[\bar{\mathbf{M}}_\mu]_{(\bar{k}-1)\bar{N}+i, (\bar{k}-1)\bar{N}+j} := \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} \mu_{knj} \mu_{kni} \, ds \quad (26)$$

for $i, j \in \{1, 2, \dots, \bar{N}\}$ and $\bar{k} \in \{1, 2, \dots, \bar{K}\}$. This leads to a block diagonal matrix because the ansatz and test functions $\mu_{kni} = \mu_{\bar{k}i}$, $i \in \{1, 2, \dots, \bar{N}\}$ have support only on the corresponding edge $E_{kn} = E_{\bar{k}}$. Term IX is very similar to term IV, where each entry of the resulting matrix $\mathbf{T} \in \mathbb{R}^{\bar{K}\bar{N} \times KN}$ is given as

$$[\mathbf{T}]_{(\bar{k}-1)\bar{N}+i, (k-1)N+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{int}}} \int_{E_{kn}} \varphi_{kj} \mu_{kni} \, ds \quad (27a)$$

with $i \in \{1, \dots, \bar{N}\}$, $j \in \{1, \dots, N\}$, and \bar{k} given by the mapping in (5). The contribution of a single edge is

$$\mathbf{T}_{E_{kn}} = \int_{E_{kn}} \begin{bmatrix} \varphi_{k1} \mu_{kn1} & \dots & \varphi_{kN} \mu_{kn1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1} \mu_{kn1} & \dots & \varphi_{kN} \mu_{kn\bar{N}} \end{bmatrix} ds. \quad (27b)$$

In fact, we have $\mathbf{T} = \mathbf{R}_\mu^T$ from (22b). Term X gives us the edge mass matrix on boundary edges

$$[\tilde{\mathbf{M}}_\mu]_{(\bar{k}-1)\bar{N}+i, (\bar{k}-1)\bar{N}+j} := \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{ext}}} \int_{E_{kn}} \mu_{knj} \mu_{kni} \, ds \quad (28)$$

for $i, j \in \{1, 2, \dots, \bar{N}\}$, $\bar{k} \in \{1, 2, \dots, \bar{K}\}$ and the matrix entries given in (26). These integrals are over edges on the domain boundary, so that each integral is only evaluated once.

The last term — term XI — incorporates boundary data into the hybrid equation. For the inflow boundary edges, we obtain a contribution to the right-hand-side $\mathbf{K}_{\mu, \text{in}} \in \mathbb{R}^{\bar{K}\bar{N}}$

$$[\mathbf{K}_{\mu, \text{in}}]_{(\bar{k}-1)\bar{N}+i} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{in}}} \int_{E_{kn}} c_D \mu_{kni} \, ds, \quad (29a)$$

and outflow boundaries add a contribution to the matrix $\mathbf{K}_{\mu, \text{out}} \in \mathbb{R}^{\bar{K}\bar{N} \times KN}$

$$[\mathbf{K}_{\mu, \text{out}}]_{(\bar{k}-1)\bar{N}+i, (k-1)N+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\text{out}}} \int_{E_{kn}} \varphi_{kj} \mu_{kni} \, ds \quad (29b)$$

meaning that we obtain a block matrix similar to \mathbf{T} (cf. (27b)) from every outflow edge.

3.3. Time discretization

The system of equations in (17) can be reformulated in matrix notation as

$$\begin{aligned} \mathbf{M}_\varphi \partial_t \mathbf{C}(t) &= \mathbf{B}_\varphi(t) - \bar{\mathbf{L}}(t) \mathbf{C}(t) - \bar{\mathbf{M}}(t) \boldsymbol{\Lambda}(t) =: \mathbf{R}_{\text{RK}}(t, \mathbf{C}(t), \boldsymbol{\Lambda}(t)), \\ \mathbf{0} &= \mathbf{B}_\mu(t) - \mathbf{N}(t) \mathbf{C}(t) - \mathbf{P} \boldsymbol{\Lambda}(t). \end{aligned}$$

This is a first order differential algebraic equation [26], and we use implicit time stepping schemes to discretize it in a stable manner. Here, we employ diagonally implicit Runge-Kutta (DIRK) schemes of orders 1 to 4 [46, 47].

The time interval $[t^0, t^{\text{end}}]$ is discretized into not necessarily equidistant points t^n with $t^0 = 0 < t^1 < t^2 < \dots < t^{\text{end}}$. The time step size of the n th time step is given by $\Delta t^n := t^{n+1} - t^n$, and we abbreviate coefficient vectors and matrices on the n th time level as \mathbf{C}^n , etc. Time step adaptation can be easily achieved in DIRK schemes with embedded error estimators, e.g., as carried out for HDG methods in [48].

For a stiffly accurate DIRK scheme with s stages, the update at t^{n+1} is obtained by solving

$$\begin{aligned} \mathbf{M}_\varphi \mathbf{C}^{(i)} &= \mathbf{M}_\varphi \mathbf{C}^n + \Delta t \sum_{j=1}^i a_{ij} \mathbf{R}_{\text{RK}}(t^{(j)}, \mathbf{C}^{(j)}, \boldsymbol{\Lambda}^{(j)}) \quad i = 1, \dots, s, \\ \mathbf{0} &= \mathbf{B}_\mu^{(i)} - \mathbf{N}^{(i)} \mathbf{C}^{(i)} - \mathbf{P} \boldsymbol{\Lambda}^{(i)} \end{aligned}$$

c_1	a_{11}		
\vdots	\vdots	\ddots	
c_s	a_{s1}	\cdots	a_{ss}
	b_1	\cdots	b_s

Table 1: Butcher tableau of a diagonally implicit Runge-Kutta method. Coefficients of the upper triangular part are zeros.

and setting

$$\mathbf{C}^{n+1} = \mathbf{C}^{(s)}$$

with $t^{(i)} = t^n + c_i \Delta t^n$ and coefficients a_{ij}, b_j, c_i defined in the routine `rungeKuttaImplicit` (see Butcher tableau in Table 1). Due to the implicit character of the time iteration, each stage of the DIRK method requires solution of a linear equation system. This is where the hybridization comes in handy: it reduces the size of the system that has to be solved. More details on this are given in Sec. 3.4.

Remark 1. *All of the employed DIRK schemes are A- and L-stable [47].*

Remark 2. *The DIRK schemes are stiffly accurate, i.e.*

$$b_j = a_{sj}, \quad j = 1, \dots, s.$$

This means in particular that the last update $\mathbf{C}^{(s)}$ is actually the updated solution at the new time t^{n+1} .

3.4. Static condensation

In the i th stage of a DIRK scheme, one has to solve the following system of equations:

$$\underbrace{\left(\mathbf{M}_\varphi + a_{ii} \Delta t \bar{\mathbf{L}}^{(i)} \right)}_{=\mathbf{L}} \mathbf{C}^{(i)} + \underbrace{a_{ii} \Delta t \bar{\mathbf{M}}^{(i)}}_{=\mathbf{M}} \boldsymbol{\Lambda}^{(i)} = \underbrace{\mathbf{M}_\varphi \mathbf{C}^n + a_{ii} \Delta t \mathbf{B}_\varphi^{(i)} + \Delta t \sum_{j=1}^{i-1} a_{ij} \mathbf{R}_{\text{RK}}(t^{(j)}, \mathbf{C}^{(j)}, \boldsymbol{\Lambda}^{(j)})}_{=\mathbf{Q}},$$

$$\mathbf{N}^{(i)} \mathbf{C}^{(i)} + \mathbf{P} \boldsymbol{\Lambda}^{(i)} = \mathbf{B}_\mu^{(i)}.$$

We can write this compactly as

$$\mathbf{L} \mathbf{C}^{(i)} + \mathbf{M} \boldsymbol{\Lambda}^{(i)} = \mathbf{Q}, \tag{30a}$$

$$\mathbf{N} \mathbf{C}^{(i)} + \mathbf{P} \boldsymbol{\Lambda}^{(i)} = \mathbf{B}_\mu^{(i)}. \tag{30b}$$

The first line (30a) refers to equation (17a), and the second line (30b) to the hybrid part (17b). Now, we want to demonstrate the solution procedure in detail: Unhybridized DG methods would require a matrix of the same size as $\mathbf{L} \in \mathbb{R}^{KN \times KN}$ to be inverted implying rapid growth of the matrix size for high polynomial orders since $N = \frac{(p+1)(p+2)}{2} = \mathcal{O}(p^2)$ in 2D. This is especially pronounced in comparison to other methods such as continuous finite elements, where the continuity requirements in the discrete space definition reduce the number of degrees of freedom. Using Schur complement reduction procedure, we obtain by substitution

$$\mathbf{C}^{(i)} = \mathbf{L}^{-1}(\mathbf{Q} - \mathbf{M} \boldsymbol{\Lambda}^{(i)}) \tag{31}$$

$$(-\mathbf{N} \mathbf{L}^{-1} \mathbf{M} + \mathbf{P}) \boldsymbol{\Lambda}^{(i)} = \mathbf{B}_\mu^{(i)} - \mathbf{N} \mathbf{L}^{-1} \mathbf{Q}. \tag{32}$$

This leads to a memory (and time) efficient procedure to solve this system. First, we invert \mathbf{L} to compute $\mathbf{L}^{-1} \mathbf{Q}$ and $\mathbf{L}^{-1} \mathbf{M}$, which can be done in a local fashion because \mathbf{L} is a block diagonal matrix. Therefore, $\mathbf{L}^{-1} \mathbf{Q}$ and $\mathbf{L}^{-1} \mathbf{M}$ are often

referred to as local solves. Then, we construct (32) using $\mathbf{L}^{-1}\mathbf{Q}$ and $\mathbf{L}^{-1}\mathbf{M}$ and solve for $\Lambda^{(i)}$. Once $\Lambda^{(i)}$ is known, we can update $\mathbf{C}^{(i)}$ by substituting the updated value of $\Lambda^{(i)}$ into (31).

The local solves can be implemented in several different ways: In our MATLAB / GNU Octave implementation, it turned out to be most efficient to explicitly compute \mathbf{L}^{-1} in a block-wise fashion and then to apply the inverse matrix to \mathbf{Q} and \mathbf{M} . Thus we select a number of elements and invert the corresponding blocks at once instead of inverting all element-blocks separately or inverting the entire matrix at once. This block-wise inversion is implemented in the routine `blkinv`. The optimal block size depends on the utilized hardware especially on the cache sizes of the employed CPU. Heuristically, we determined $32 \cdot 2^{-p} \cdot N$ to be a good choice in our case (for hardware details see Table 5). Optionally, one could parallelize the local solves since they do not depend on each other.

4. Implementation

A description of data structures and algorithms related to meshing can be found in the first paper in series [34]. For the sake of completeness, we briefly introduce transformation rules for element and edge integrals that are detailed in previous works [34, 35] and emphasize differences due to edge unknown λ_h . Finally, we discuss the assembly of the block matrices in Sec. 4.3.

4.1. Backtransformation to reference element and reference interval

We use an affine mapping from the reference triangle $\hat{T} = \{[0, 0]^T, [1, 0]^T, [0, 1]^T\}$ to any $T_k = \{\mathbf{x}_{k1}, \mathbf{x}_{k2}, \mathbf{x}_{k3}\} \in \mathcal{T}_h$,

$$\mathbf{F}_k : \hat{T} \ni \hat{\mathbf{x}} \mapsto \mathbf{B}_k \hat{\mathbf{x}} + \mathbf{x}_{k1} = \mathbf{x} \in T_k, \quad \text{with} \quad \mathbb{R}^{2 \times 2} \ni \mathbf{B}_k := [\mathbf{x}_{k2} - \mathbf{x}_{k1} \mid \mathbf{x}_{k3} - \mathbf{x}_{k1}]. \quad (33)$$

It holds $0 < \det \mathbf{B}_k = 2|T_k|$, and thus the component-wise definition of the mapping and its inverse read as

$$\mathbf{F}_k(\hat{\mathbf{x}}) = \begin{bmatrix} B_k^{11} \hat{x}^1 + B_k^{12} \hat{x}^2 + x_{k1}^1 \\ B_k^{21} \hat{x}^1 + B_k^{22} \hat{x}^2 + x_{k1}^2 \end{bmatrix} \quad \text{and} \quad \mathbf{F}_k^{-1}(\mathbf{x}) = \frac{1}{2|T_k|} \begin{bmatrix} B_k^{22}(x^1 - x_{k1}^1) - B_k^{12}(x^2 - x_{k1}^2) \\ B_k^{11}(x^2 - x_{k1}^2) - B_k^{21}(x^1 - x_{k1}^1) \end{bmatrix}.$$

For functions $w : T_k \rightarrow \mathbb{R}$ and $\hat{w} : \hat{T} \rightarrow \mathbb{R}$, we imply $\hat{w} = w \circ \mathbf{F}_k$, i. e., $w(\mathbf{x}) = \hat{w}(\hat{\mathbf{x}})$. The gradient is transformed using the chain rule:

$$\nabla = (\hat{\nabla} \mathbf{F}_k)^{-T} \hat{\nabla} = \frac{1}{2|T_k|} \begin{bmatrix} B_k^{22} \partial_{\hat{x}^1} - B_k^{21} \partial_{\hat{x}^2} \\ B_k^{11} \partial_{\hat{x}^2} - B_k^{12} \partial_{\hat{x}^1} \end{bmatrix}, \quad (34)$$

where we abbreviated $\hat{\nabla} = [\partial_{\hat{x}^1}, \partial_{\hat{x}^2}]^T$. This results in transformation formulas for integrals over an element T_k or an edge $E_{kn} \subset T_k$ for a function $w : \Omega \rightarrow \mathbb{R}$

$$\int_{T_k} w(\mathbf{x}) d\mathbf{x} = \frac{|T_k|}{|\hat{T}|} \int_{\hat{T}} w \circ \mathbf{F}_k(\hat{\mathbf{x}}) d\hat{\mathbf{x}} = 2|T_k| \int_{\hat{T}} w \circ \mathbf{F}_k(\hat{\mathbf{x}}) d\hat{\mathbf{x}} = 2|T_k| \int_{\hat{T}} \hat{w}(\hat{\mathbf{x}}) d\hat{\mathbf{x}}, \quad (35a)$$

$$\int_{E_{kn}} w(\mathbf{x}) d\mathbf{x} = \frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} w \circ \mathbf{F}_k(\hat{\mathbf{x}}) d\hat{\mathbf{x}} = \frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} \hat{w}(\hat{\mathbf{x}}) d\hat{\mathbf{x}}. \quad (35b)$$

We introduce the mapping $\hat{\gamma}_n : [0, 1] \rightarrow \hat{E}_n$ from the reference interval $[0, 1]$ to the n th edge of the reference element, which is given by

$$\hat{\gamma}_1(s) = \begin{bmatrix} 1-s \\ s \end{bmatrix}, \quad \hat{\gamma}_2(s) = \begin{bmatrix} 0 \\ 1-s \end{bmatrix}, \quad \hat{\gamma}_3(s) = \begin{bmatrix} s \\ 0 \end{bmatrix}, \quad (36)$$

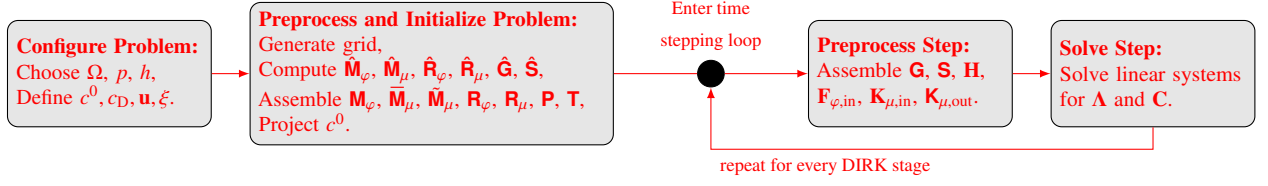


Figure 2: Structure of the solution algorithm. The last two steps are executed repeatedly as part of the time stepping loop.

and use this to transform Eq. (35b) further

$$\frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} \hat{w}(\hat{\mathbf{x}}) d\hat{\mathbf{x}} = \frac{|E_{kn}|}{|\hat{E}_n|} \int_0^1 \hat{w} \circ \hat{\gamma}_n(s) |\hat{\gamma}'_n(s)| ds = |E_{kn}| \int_0^1 \hat{w} \circ \hat{\gamma}_n(s) ds, \quad (37)$$

where we use the fact that $|\hat{\gamma}'_n(s)| = |\hat{E}_n|$.

A difference compared to previous publications in series are edge integrals with basis functions from an adjoining element and basis functions defined on the edge, e.g., $\int_{E_{kn}} \varphi_{ki} \mu_{\bar{k}j} d\mathbf{x}$ with $\bar{k} = \rho(k, n)$ as defined in (5). They are transformed according to transformation rules (35b), (37)

$$\int_{E_{kn}} \varphi_{ki} \mu_{\bar{k}j} d\mathbf{x} = \frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_k} \hat{\varphi}_i(\hat{\mathbf{x}}) \mu_{knj} \circ \mathbf{F}_k(\hat{\mathbf{x}}) d\hat{\mathbf{x}} = |E_{kn}| \int_0^1 \hat{\varphi}_i \circ \hat{\gamma}_n(s) \hat{\mu}_j \circ \hat{\beta}_{kn}(s) ds, \quad (38)$$

where we introduced an additional mapping $\hat{\beta}_{kn} : [0, 1] \rightarrow [0, 1]$ that adapts the edge orientation to match the definition of $\mu_{knj} = \mu_{\bar{k}j}$ and is defined as

$$\hat{\beta}_{kn}(s) = \begin{cases} s & \text{if } \kappa(\rho(k, n), 1) = k, \\ 1 - s & \text{if } \kappa(\rho(k, n), 2) = k \end{cases} \quad (39)$$

with $\kappa(\rho(k, n), l) = \kappa(\bar{k}, l)$ given in (6). This does not introduce any further terms into the equation since $|\hat{\beta}'_{kn}(s)| = 1$.

Similarly to other publications in this series [34, 35], triangle and edge integrals are approximated using numerical quadrature rules after transformation to reference element \hat{T} or reference interval $[0, 1]$, respectively. We abstain from reproducing the details here and refer the interested reader to previous works. In all numerical experiments, we use quadrature rules of order $2p + 1$ on both elements and edges.

4.2. Program structure

The implemented solver follows the structure outlined in Fig. 2. It consists of a pre-processing phase that includes the definition of solver parameters and assembly of stationary matrix blocks and global matrices. In the time stepping loop, the time-dependent global matrices and right hand side vectors are assembled before solving the resulting linear system. Note that this has to be done for every stage of the DIRK schemes (cf. Sec. 3.3).

4.3. Assembly

In this section, the vectorized assembly of the block matrices in (17) is outlined. For that purpose, the required terms are transformed to reference triangle \hat{T} or reference interval $[0, 1]$ and then evaluated by numerical quadrature.

As in previous papers in series [34, 35], we make extensive use of the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{R}^{m_a \times n_a}$, $\mathbf{B} \in \mathbb{R}^{m_b \times n_b}$ defined as

$$\mathbf{A} \otimes \mathbf{B} := \left[[\mathbf{A}]_{i,j} \mathbf{B} \right] \in \mathbb{R}^{m_a m_b \times n_a n_b}. \quad (40)$$

Additionally, we employ operation $\mathbf{A} \otimes_{\vee} \mathbf{B}$ with $m_b = r m_a, r \in \mathbb{N}$ introduced in our last publication [35] as

$$\mathbf{A} \otimes_{\vee} \mathbf{B} := \left[[\mathbf{A}]_{i,j} [\mathbf{B}]_{(i-1)r;ir,:} \right] \in \mathbb{R}^{m_b \times n_a n_b}, \quad (41)$$

which can be interpreted as a Kronecker product that takes a different right-hand-side for every row of the left-hand side. This operation is implemented in the routine `kronVec`. In many cases, we must select edges matching a certain criterion, e.g., edges in the interior $E_{kn} \in \mathcal{E}_{\text{int}}$. We denote this using the Kronecker delta symbol with a matching subscript that indicates the criterion to be met, for example

$$\delta_{E_{kn} \in \mathcal{E}_{\text{int}}} := \begin{cases} 1 & \text{if } E_{kn} \in \mathcal{E}_{\text{int}}, \\ 0 & \text{otherwise.} \end{cases}$$

Some of the block-matrices in system (17) appeared in identical or only slightly different form in previous publications [34, 35]. For brevity, we abstain from reproducing the corresponding assembly steps here and refer to the existing descriptions. This is the case for matrix \mathbf{R}_{φ} (identical to \mathbf{S}^{diag} in [34]), source terms \mathbf{H} (cf. any of the papers in series), and mass matrix \mathbf{M}_{φ} . Vector $\mathbf{F}_{\varphi, \text{in}}$ in equation (17a) contains Dirichlet data provided on inflow boundary edges and is very similar to vector \mathbf{K}_D in [35] only lacking the upwinding per quadrature point. However, here we do not evaluate \mathbf{u} in all quadrature points during the assembly of the vector but once per time level and use it then for other terms as well (see Sec. 4.3.2). We do the same for Dirichlet data c_D and pass them together as per-quadrature-point values to the assembly routine `assembleVecEdgePhiIntVal`, which is the same as `assembleVecEdgePhiIntFuncContVal` in [35] without the evaluation of the coefficient function.

Differences to the standard DG implementation in [35] arise in all matrices that stem from edge integrals involving the new unknown λ (see Sec. 4.3.2–4.3.5) and in matrices \mathbf{G}^m (see Sec. 4.3.1) that are assembled without projecting $\mathbf{u}(t, \mathbf{x})$ into the broken polynomial space.

4.3.1. Assembly of \mathbf{G}^m

For the assembly of matrices \mathbf{G}^m from (19), we make use of the transformation rule for the gradient (34). Due to the time-dependent function $u^m(t, \mathbf{x})$ in the integrand, we cannot reduce the assembly to Kronecker products of reference matrices as we did for the mass matrix. We apply transformation rules (34), (35a) and obtain

$$\begin{aligned} \int_{T_k} u^1(t, \mathbf{x}) \varphi_{kj} \partial_{x^1} \varphi_{ki} d\mathbf{x} &\approx \sum_{r=1}^R \left(\mathbf{B}_k^{22} [\mathbf{U}^1]_{k,r} [\hat{\mathbf{G}}]_{1,r,i,j} - \mathbf{B}_k^{21} [\mathbf{U}^1]_{k,r} [\hat{\mathbf{G}}]_{2,i,j,r} \right), \\ \int_{T_k} u^2(t, \mathbf{x}) \varphi_{kj} \partial_{x^2} \varphi_{ki} d\mathbf{x} &\approx \sum_{r=1}^R \left(-\mathbf{B}_k^{12} [\mathbf{U}^2]_{k,r} [\hat{\mathbf{G}}]_{1,r,i,j} + \mathbf{B}_k^{11} [\mathbf{U}^2]_{k,r} [\hat{\mathbf{G}}]_{2,i,j,r} \right) \end{aligned}$$

with multidimensional array $\hat{\mathbf{G}} \in \mathbb{R}^{2 \times N \times N \times R}$ that represents a part of the contribution of the quadrature rule in every integration point $\hat{\mathbf{q}}_r$ on the reference element \hat{T} and arrays $\mathbf{U}^m \in \mathbb{R}^{K \times R}$ that hold the velocity components evaluated in each quadrature point of each element

$$[\hat{\mathbf{G}}]_{m,i,j,r} := \omega_r \partial_{\hat{x}^m} \hat{\varphi}_{ki} \hat{\varphi}_{kj}, \quad [\mathbf{U}^m]_{k,r} := u^m(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)).$$

The element-local matrix $\mathbf{G}_{T_k}^1 \in \mathbb{R}^{N \times N}$ is then given as

$$\begin{aligned} \mathbf{G}_{T_k}^1 &= \sum_{r=1}^R \left(B_k^{22} \begin{bmatrix} \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^1} \hat{\varphi}_1 \hat{\varphi}_1 & \cdots & \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^1} \hat{\varphi}_1 \hat{\varphi}_N \\ \vdots & \ddots & \vdots \\ \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^1} \hat{\varphi}_N \hat{\varphi}_1 & \cdots & \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^1} \hat{\varphi}_N \hat{\varphi}_N \end{bmatrix} \right. \\ &\quad \left. - B_k^{21} \begin{bmatrix} \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^2} \hat{\varphi}_1 \hat{\varphi}_1 & \cdots & \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^2} \hat{\varphi}_1 \hat{\varphi}_N \\ \vdots & \ddots & \vdots \\ \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^2} \hat{\varphi}_N \hat{\varphi}_1 & \cdots & \omega_r u^1(t, \mathbf{F}_k(\hat{\mathbf{q}}_r)) \partial_{x^2} \hat{\varphi}_N \hat{\varphi}_N \end{bmatrix} \right) \\ &= \sum_{r=1}^R \left(B_k^{22} [\mathbf{U}^1]_{k,r} [\hat{\mathbf{G}}]_{1,::,r} - B_k^{21} [\mathbf{U}^1]_{k,r} [\hat{\mathbf{G}}]_{2,::,r} \right). \end{aligned}$$

On the surface, this procedure appears to closely follow the assembly of \mathbf{G}^m in [35]; however, conceptually, there is a big difference: here a quadrature rule is applied to all elements at the same time using blocks of basis functions evaluated in each quadrature point, whereas the procedure to assemble \mathbf{G}^m in [35] builds the full matrix from the contributions of each degree of freedom of the projected DG representation of the velocity using already integrated reference blocks. To speed up the implementation, we do not assemble a global sparse matrix in each iteration of the `for`-loop over quadrature points and adding to the sparse matrix from the previous iteration. Instead, we use the standard Kronecker product (40) to build a dense $KN \times N$ vector of blocks, from which the global sparse matrix is constructed using the vectorial Kronecker operator (41)

$$\mathbf{G}^1 = \sum_{r=1}^R \mathbf{I}_{K \times K} \otimes_{\mathbf{V}} \left(\begin{bmatrix} B_1^{22} [\mathbf{U}^1]_{1,r} \\ \vdots \\ B_K^{22} [\mathbf{U}^1]_{K,r} \end{bmatrix} \otimes [\hat{\mathbf{G}}]_{1,::,r} - \begin{bmatrix} B_1^{21} [\mathbf{U}^1]_{1,r} \\ \vdots \\ B_K^{21} [\mathbf{U}^1]_{K,r} \end{bmatrix} \otimes [\hat{\mathbf{G}}]_{2,::,r} \right),$$

where $\mathbf{I}_{K \times K}$ is the $K \times K$ identity matrix. \mathbf{G}^2 is assembled analogously. The procedure for both matrices is implemented in the routine `assembleMatElemDphiPhiFuncContVec`.

4.3.2. Assembly of \mathbf{S} and \mathbf{S}_{out}

Matrices \mathbf{S} and \mathbf{S}_{out} are assembled together, the set of relevant edges is expanded to $\mathcal{E}_{int} \cup \mathcal{E}_{out}$. On a relevant edge E_{kn} , we transform terms of the form given in Eq. (21a) using transformation rule (38) and approximate the integral by a one-dimensional numerical quadrature rule

$$\begin{aligned} \int_{E_{kn}} (\mathbf{u} \cdot \mathbf{v}_{kn}) \varphi_{ki} \mu_{knj} \, ds &= |E_{kn}| \int_0^1 ((\mathbf{u}(t) \circ \mathbf{F}_k \circ \hat{\gamma}_n(s)) \cdot \mathbf{v}_{kn}) \hat{\varphi}_i \circ \hat{\gamma}_n(s) \hat{\mu}_j \circ \hat{\beta}_{kn}(s) \, ds \\ &\approx \sum_{r=1}^R \underbrace{((\mathbf{u}(t) \circ \mathbf{F}_k \circ \hat{\gamma}_n(\hat{q}_r)) \cdot \mathbf{v}_{kn})}_{=:[\mathbf{U}_v]_{k,n,r}} \underbrace{\omega_r \hat{\varphi}_i \circ \hat{\gamma}_n(\hat{q}_r) \hat{\mu}_j \circ \hat{\beta}_{kn}(\hat{q}_r)}_{=:[\hat{\mathbf{S}}]_{i,j,n,r,l}}, \end{aligned}$$

where $\mathbf{U}_v \in \mathbb{R}^{K \times 3 \times R}$ holds the normal velocity evaluated in each quadrature point, and the subscript l in $\hat{\mathbf{S}} \in \mathbb{R}^{N \times \bar{N} \times 3 \times R \times 2}$ covers the two cases of $\hat{\beta}_{kn}$ in (39). This allows us to assemble the global matrix as

$$\mathbf{S} = \sum_{n=1}^3 \underbrace{\begin{bmatrix} \delta_{E_{1n}=E_1} & \cdots & \delta_{E_{1n}=E_{\bar{K}}} \\ \vdots & \ddots & \vdots \\ \delta_{E_{Kn}=E_1} & \cdots & \delta_{E_{Kn}=E_{\bar{K}}} \end{bmatrix}}_{=\mathbf{\Delta}_n} \otimes_{\mathbf{V}} \left(\sum_{r=1}^R \sum_{l=1}^2 \begin{bmatrix} \delta_{E_{1n} \in \mathcal{E}_{int} \cup \mathcal{E}_{out}} |E_{1n}| \delta_{\kappa(\rho(1,n),l)=1} [\mathbf{U}_v]_{1,n,r} \\ \vdots \\ \delta_{E_{Kn} \in \mathcal{E}_{int} \cup \mathcal{E}_{out}} |E_{Kn}| \delta_{\kappa(\rho(K,n),l)=K} [\mathbf{U}_v]_{1,n,r} \end{bmatrix} \otimes [\hat{\mathbf{S}}]_{:,::,n,r,l} \right). \quad (42)$$

Here we introduce the permutation matrix $\mathbf{\Delta}_n \in \mathbb{R}^{K \times \bar{K}}$, $n \in \{1, 2, 3\}$ that has a single entry per row indicating the correspondence $E_{kn} = E_{\bar{k}}$ for all elements and edges. It takes care of the necessary permutation from the element-based view of the assembly to the edge-based view of the edge degrees of freedom. The assembly of \mathbf{S} is implemented in the routine `assembleMatEdgePhiIntMuVal`.

4.3.3. Assembly of \mathbf{R}_μ , \mathbf{T} , and $\mathbf{K}_{\mu,\text{out}}$

Matrices \mathbf{R}_μ , \mathbf{T} , and $\mathbf{K}_{\mu,\text{out}}$ are all constructed using similar terms with the only difference being the set of edges considered or the roles of φ and μ interchanged. The terms in Eqs. (22a), (27a), or (29b) are transformed using transformation rule (38) yielding

$$\int_{E_{kn}} \varphi_{ki} \mu_{knj} \, ds = |E_{kn}| \underbrace{\int_0^1 \hat{\varphi}_i \circ \hat{\gamma}_n(s) \hat{\mu}_j \circ \hat{\beta}_{kn}(s) \, ds}_{=:[\hat{\mathbf{R}}_\mu]_{i,j,n,l}},$$

where the index l in $\hat{\mathbf{R}}_\mu \in \mathbb{R}^{N \times \bar{N} \times 3 \times 2}$ plays the same role as it did for $\hat{\mathbf{S}}$ before covering the two cases of $\hat{\beta}_{kn}$. Thus we can assemble \mathbf{R}_μ and \mathbf{T} as

$$\mathbf{R}_\mu = \sum_{n=1}^3 \sum_{l=1}^2 \left(\begin{bmatrix} |E_{1n}| \delta_{E_{1n} \in \mathcal{E}_{\text{int}}} & & \\ & \ddots & \\ & & |E_{Kn}| \delta_{E_{Kn} \in \mathcal{E}_{\text{int}}} \end{bmatrix} \mathbf{\Delta}_n \right) \otimes [\hat{\mathbf{R}}_\mu]_{:,:,n,l} = \mathbf{T}^T$$

with $\mathbf{\Delta}_n$ from Eq. (42). These matrices are time-independent and thus assembled only once in `preprocessProblem`, using routine `assembleMatEdgePhiIntMu`.

Matrix $\mathbf{K}_{\mu,\text{out}}$ only differs in the set of edges considered, hence it can be assembled almost identically as

$$(\mathbf{K}_{\mu,\text{out}})^T = \sum_{n=1}^3 \sum_{l=1}^2 \left(\begin{bmatrix} |E_{1n}| \delta_{E_{1n} \in \mathcal{E}_{\text{out}}} & & \\ & \ddots & \\ & & |E_{Kn}| \delta_{E_{Kn} \in \mathcal{E}_{\text{out}}} \end{bmatrix} \mathbf{\Delta}_n \right) \otimes [\hat{\mathbf{R}}_\mu]_{:,:,n,l}.$$

Note that, in fact, we assemble the transpose of $\mathbf{K}_{\mu,\text{out}}$ and thus can reuse the same function. However, due to the time-dependent velocity field, the set of outflow edges \mathcal{E}_{out} can change over time, and we have to do this in every stage of the time stepping method.

4.3.4. Assembly of $\bar{\mathbf{M}}_\mu$ and $\tilde{\mathbf{M}}_\mu$

For the hybrid mass matrices $\bar{\mathbf{M}}_\mu$ and $\tilde{\mathbf{M}}_\mu$ (cf. Eq. (26) and (28)), we apply transformation rule (38) to obtain

$$\int_{E_{kn}} \mu_{knj} \mu_{kni} \, ds = |E_{kn}| \int_0^1 \hat{\mu}_j \circ \hat{\beta}_{kn}(s) \hat{\mu}_i \circ \hat{\beta}_{kn}(s) \, ds = |E_{kn}| \underbrace{\int_0^1 \hat{\mu}_j(s) \hat{\mu}_i(s) \, ds}_{=:[\hat{\mathbf{M}}_\mu]_{i,j}}.$$

With the help of permutation matrices $\mathbf{\Delta}_n$ (cf. Eq. (42)), we obtain the global matrices

$$\begin{aligned} \bar{\mathbf{M}}_\mu &= \sum_{n=1}^3 \left((\mathbf{\Delta}_n)^T \begin{bmatrix} |E_{1n}| \delta_{E_{1n} \in \mathcal{E}_{\text{int}}} & & \\ & \ddots & \\ & & |E_{Kn}| \delta_{E_{Kn} \in \mathcal{E}_{\text{int}}} \end{bmatrix} \mathbf{\Delta}_n \right) \otimes \hat{\mathbf{M}}_\mu, \\ \tilde{\mathbf{M}}_\mu &= \sum_{n=1}^3 \left((\mathbf{\Delta}_n)^T \begin{bmatrix} |E_{1n}| \delta_{E_{1n} \in \mathcal{E}_{\text{ext}}} & & \\ & \ddots & \\ & & |E_{Kn}| \delta_{E_{Kn} \in \mathcal{E}_{\text{ext}}} \end{bmatrix} \mathbf{\Delta}_n \right) \otimes \hat{\mathbf{M}}_\mu, \end{aligned}$$

which are implemented in a common assembly routine `assembleMatEdgeMuMu`.

4.3.5. Assembly of $\mathbf{K}_{\mu,\text{in}}$

Last, we consider the Dirichlet boundary contributions on inflow boundary edges in the hybrid equation. We transform the term in Eq. (4.3.5) as before and approximate it using numerical quadrature

$$\int_{E_{kn}} c_D \mu_{kni} \, ds = |E_{kn}| \int_0^1 c_D(t) \circ \mathbf{F}_k \circ \hat{\gamma}_n(s) \hat{\mu}_i(s) \, ds \approx |E_{kn}| \sum_{r=1}^R \omega_r c_D(t) \circ \mathbf{F}_k \circ \hat{\gamma}_n(\hat{q}_r) \hat{\mu}_i(\hat{q}_r).$$

We can omit the mapping $\hat{\beta}_{kn}$ here, since we only consider boundary edges, and our numbering of the mesh entities ensures that only the first case of the definition of $\hat{\beta}_{kn}$ (cf. Eq. (39)) is relevant here. Thus the global vector is assembled as

$$\mathbf{K}_{\mu,\text{in}} = \sum_{n=1}^3 |E_{kn}| \sum_{r=1}^R \omega_r \hat{\mu}_i(\hat{q}_r) (\mathbf{\Delta}_n)^T \begin{bmatrix} \delta_{E_{1n} \in \mathcal{E}_{\text{in}}} & & \\ & \ddots & \\ & & \delta_{E_{Kn} \in \mathcal{E}_{\text{in}}} \end{bmatrix} [\mathbf{C}_D]_{:,n,r},$$

which is implemented in `assembleVecEdgeMuFuncCont`.

5. Numerical results

In this section, we verify our implementation by means of convergence experiments followed by some performance analysis of the code. In addition, a runtime comparison of the presented HDG discretization to a time-implicit version of the DG discretization from the previous publication [35] is given to illustrate some points mentioned in this work.

5.1. Analytical convergence tests

Our implementation is verified by comparing the experimental orders of convergence to the analytically predicted ones for smooth solutions. For that, we choose an exact solution $c(t, \mathbf{x})$ and a velocity field $\mathbf{u}(t, \mathbf{x})$, with which we derive boundary data c_D and source term ξ analytically by substituting c and \mathbf{u} into (1)–(2). The discretization error $\|c_h - c\|_{L^2(\Omega)}$ is computed as the L^2 -norm of the difference between the numerical and the analytical solutions at the end time (cf. [34]). From that, the experimental orders of convergence EOC is given by

$$\text{EOC} := \ln \left(\frac{\|c_{h_{j-1}} - c\|_{L^2(\Omega)}}{\|c_{h_j} - c\|_{L^2(\Omega)}} \right) \bigg/ \ln \left(\frac{h_{j-1}}{h_j} \right).$$

5.1.1. Steady problem

p	0		1		2		3		4	
j	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC
1	2.69e-01	—	6.42e-02	—	7.35e-03	—	1.50e-03	—	1.87e-04	—
2	1.86e-01	0.53	1.75e-02	1.88	7.41e-04	3.31	9.79e-05	3.94	6.16e-06	4.92
3	1.18e-01	0.66	4.32e-03	2.02	8.55e-05	3.12	6.26e-06	3.97	1.95e-07	4.98
4	6.87e-02	0.78	1.07e-03	2.01	1.04e-05	3.03	3.95e-07	3.99	6.11e-09	4.99
5	3.78e-02	0.86	2.68e-04	2.00	1.30e-06	3.01	2.48e-08	3.99	1.92e-10	5.00
6	2.00e-02	0.92	6.71e-05	2.00	1.62e-07	3.00	1.55e-09	4.00	6.00e-12	5.00

Table 2: $L^2(\Omega)$ discretization errors for the steady problem in Sec. 5.1.1 and experimental orders of convergence for different polynomial degrees. We have $h_j = \frac{1}{3 \cdot 2^j}$ and $K = 18 \cdot 4^j$ triangles in the j th refinement level.

To verify our spatial discretization we choose the exact solution $c(\mathbf{x}) := \cos(7x^1) \cos(7x^2)$ and velocity field $\mathbf{u}(\mathbf{x}) := [\exp((x^1 + x^2)/2), \exp(x^1 - x^2)/2)]^T$ on the unit square $(0, 1)^2$. We omit the time discretization and solve the problem directly for c_{h_j} on a sequence of increasingly finer meshes with element sizes $h_j = \frac{1}{3 \cdot 2^j}$ yielding the expected orders of convergence $\text{EOC} = p + 1$ for $p > 0$ as listed in Table 2.

5.1.2. Unsteady problem (ODE)

p	0		1		2		3		4	
j	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC
1	4.25e-02	—	8.30e-05	—	6.79e-06	—	1.13e-08	—	1.13e-08	—
2	2.14e-02	0.99	2.13e-05	1.96	8.53e-07	3.00	7.20e-10	3.97	7.20e-10	3.97
3	1.08e-02	0.99	5.40e-06	1.98	1.07e-07	3.00	4.54e-11	3.99	4.54e-11	3.99
4	5.39e-03	1.00	1.36e-06	1.99	1.34e-08	3.00	2.85e-12	3.99	2.86e-12	3.99
5	2.70e-03	1.00	3.40e-07	2.00	1.67e-09	3.00	2.10e-13	3.76	2.43e-13	3.56

Table 3: $L^2(\Omega)$ discretization errors for the unsteady problem in Sec. 5.1.2 measured at $t_{\text{end}} = 2$ and experimental orders of convergence for different polynomial degrees. We have $h_j = \frac{1}{3 \cdot 2^j}$, $K = 4608$, and $\Delta t_j = \frac{1}{5 \cdot 2^j}$ in the j th refinement level.

We test our implementation of the DIRK schemes (cf. Sec. 3.3) using exact solution $c(t, \mathbf{x}) = \exp(-t)$ and velocity field $\mathbf{u}(t, \mathbf{x}) = \mathbf{0}$ on $\Omega = (0, 1)^2$ in the time interval $J = [0, 2]$. The mesh size is fixed as $h_j = \frac{1}{3 \cdot 2^j}$, and the time step size is $\Delta t_j = \frac{1}{5 \cdot 2^j}$. The order of the DIRK scheme is chosen to be $\min(p + 1, 4)$. Table 3 shows the expected orders of convergence $\text{EOC} = p + 1$ for $p < 4$.

5.1.3. Unsteady problem (PDE)

p	0		1		2		3		4	
j	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC	$\ c_h - c\ $	EOC
1	2.69e-01	—	6.42e-02	—	7.35e-03	—	1.50e-03	—	1.87e-04	—
2	1.86e-01	0.53	1.75e-02	1.88	7.41e-04	3.31	9.79e-05	3.94	6.16e-06	4.92
3	1.18e-01	0.66	4.32e-03	2.02	8.55e-05	3.12	6.26e-06	3.97	1.96e-07	4.97
4	6.86e-02	0.78	1.07e-03	2.01	1.04e-05	3.03	3.96e-07	3.98	7.44e-09	4.72
5	3.78e-02	0.86	2.68e-04	2.00	1.30e-06	3.01	2.52e-08	3.97	7.74e-10	3.27

Table 4: $L^2(\Omega)$ discretization errors for the unsteady problem in Sec. 5.1.3 measured at $t_{\text{end}} = 2$ and experimental orders of convergence for different polynomial degrees. We have $h_j = \frac{1}{3 \cdot 2^j}$, $K = 18 \cdot 4^j$ triangles, and $\Delta t_j = \frac{1}{5 \cdot 2^j}$ in the j th refinement level. For $p = 4$, we reduced the time step to $\Delta t_j = \frac{1}{5 \cdot 2^{j+2}}$.

To produce a combined verification of the time and space discretization, we use exact solution given by $c(t, \mathbf{x}) = \cos(7x^1) \cos(7x^2) + \exp(-t)$ and velocity field $\mathbf{u}(\mathbf{x}) := [\exp((x^1 + x^2)/2), \exp(x^1 - x^2)/2]^T$ on $\Omega = (0, 1)^2$ in the time interval $J = [0, 2]$. Mesh size and time step are refined as $h_j = \frac{1}{3 \cdot 2^j}$ and $\Delta t_j = \frac{1}{5 \cdot 2^j}$, respectively. The order of the DIRK scheme is chosen as $\min(p + 1, 4)$, and Table 4 shows the expected orders of convergence $\text{EOC} = p + 1$ for $p > 0$.

5.2. Comparison to the unhybridized DG implementation

Our previous work in series was concerned with the same model problem (1) and the same linear flux (2); however, an *unhybridized discontinuous Galerkin* discretization with explicit strong stability preserving Runge–Kutta methods was used—in contrast to a HDG discretization and diagonally *implicit* Runge–Kutta schemes utilized in the current study. To make both implementations comparable, we implemented a variant of the DG solver that incorporates the DIRK schemes and computes the solid body rotation benchmark proposed by LeVeque [49] and used in [35] to investigate the performance of slope limiters. The initial solution consists of a slotted cylinder, a sharp cone, and a smooth hump (see Fig. 3a) placed in a square domain $\Omega = (0, 1)^2$ with velocity field $\mathbf{u}(\mathbf{x}) = [0.5 - x^2, x^1 - 0.5]^T$ producing a full counterclockwise rotation of the initial scene over time interval $J = (0, 2\pi)$. With $r = 0.0225$ and

$G(\mathbf{x}, \mathbf{x}_0) := \frac{1}{0.15} \|\mathbf{x} - \mathbf{x}_0\|_2$, we choose homogeneous Dirichlet boundary conditions $c_D = 0$ and right-hand-side function $\xi = 0$ with initial data satisfying

$$c^0(\mathbf{x}) = \left\{ \begin{array}{lll} 1 & \text{if } (x^1 - 0.5)^2 + (x^2 - 0.75)^2 \leq r & \text{(slotted cylinder)} \\ 1 - G(\mathbf{x}, [0.5, 0.25]^T) & \text{if } \wedge (x^1 \leq 0.475 \vee x^1 \geq 0.525 \vee x^2 \geq 0.85) & \text{(sharp cone)} \\ \frac{1}{4}(1 + \cos(\pi G(\mathbf{x}, [0.25, 0.5]^T))) & \text{if } (x^1 - 0.25)^2 + (x^2 - 0.5)^2 \leq r & \text{(smooth hump)} \\ 0 & \text{otherwise} & \end{array} \right\}$$

An unstructured mesh generated by MATLAB's `initmesh` with maximum element size $h = 2^{-6}$ is used resulting in 14006 elements and 320 time steps.

p	$\ c_h(t_{\text{end}}) - c^0\ _{L^2(\Omega)}$		runtime [s]		Hardware/software details	
	DG	HDG	DG	HDG		
0	1.87e-01	2.35e-01	27.1	39.8	CPU	Intel Core-i7 4790 (Haswell)
1	7.25e-02	7.78e-02	157	218		
2	5.53e-02	5.44e-02	797	717	RAM	32 GBytes
3	4.02e-02	4.13e-02	3980	3166		
4	4.16e-02	4.18e-02	10996	7199	MATLAB	R2017a

Table 5: Comparison of L^2 -errors for the solid body rotation benchmark (cf. Sec. 5.2) at end time $t_{\text{end}} = 2\pi$ (using initial data as the exact solution) and runtimes for DG and HDG solvers (left). Details of the employed hardware and software (right).

Figures 3 and 4 show the computed solution at end time $t_{\text{end}} = 2\pi$ for different polynomial degrees. We chose a color map (inspired by [50]) that emphasizes violations of the discrete maximum principle: in the range $[0, 1]$ we have a color gradient from black via white to green; values in the range $[-0.1, 0) \cup (1, 1.1]$ are colored red, and values in $(-\infty, -0.1) \cup (1.1, +\infty)$ are colored yellow. To make oscillations in the bottom range better visible, we gradually reduce the opacity from above to zero. Figure 5 presents intersection lines for $p = 3$.

Clearly, the lowest-order approximation is unusable for this kind of problem in both implementations with numerical diffusion killing off most (DG) or all (HDG) of the variability of the solution. For $p > 0$, solutions from HDG and DG are in good agreement for all approximation orders. This finding is substantiated by the intersection lines in Fig. 5 and the L^2 -errors shown in Table 5 exhibiting only minor differences between both discretizations. However, also clearly visible are severe violations of the discrete maximum principles and oscillations in the wake of cylinder and cone, which do not become less pronounced with increasing approximation order. This type of behavior can be alleviated using slope limiters as shown in our previous work [35], where a post-processing step in each time level restricted the updated solution at the vertices to the bounds provided by the mean values of the adjacent elements. Unfortunately, designing slope limiters for implicit time stepping methods is not a trivial task and lies out of scope of this work.

When comparing the runtimes for both discretizations in Table 5, it becomes clear that the static condensation outlined in Sec. 3.4 becomes advantageous especially for higher approximation orders making HDG a superior approach for time-implicit high-order discretizations.

5.3. Performance analysis

A major advantage of the hybridized DG method is the fact that the globally coupled linear equation system resulting from the discretization is relatively compact and easy to solve (see Sec. 3.4 for details). To illustrate this, we present some performance results that show the runtime distribution among the different steps of the code. We disregard pre-processing and initialization tasks as these are only performed once and consider ten time steps of the

p	$h_{\max} = 2^{-4} (K = 872)$					$h_{\max} = 2^{-6} (K = 14006)$				
	runtime	assembly step		solver step		runtime	assembly step		solver step	
		G	S	\mathbf{L}^{-1}	λ		G	S	\mathbf{L}^{-1}	λ
0	0.641	0.181 (28.2%)		0.047 (7.3%)		1.156	0.367 (31.7%)		0.617 (53.4%)	
		18.2%	8.3%	25.5%	61.7%		13.4%	19.6%	4.5%	91.9%
1	0.919	0.351 (38.2%)		0.278 (30.3%)		6.698	1.820 (27.2%)		4.674 (69.8%)	
		29.3%	17.9%	33.1%	59.0%		42.6%	29.3%	23.8%	71.2%
2	1.818	0.655 (36.0%)		0.852 (46.9%)		21.945	6.271 (28.6%)		15.491 (70.6%)	
		41.5%	21.4%	34.5%	56.0%		55.8%	31.0%	25.3%	66.1%
3	5.627	3.079 (54.7%)		2.353 (41.8%)		100.089	53.218 (53.2%)		46.688 (46.6%)	
		72.2%	15.6%	33.1%	53.7%		79.2%	17.0%	26.5%	60.2%
4	10.159	6.273 (61.7%)		3.685 (36.3%)		230.148	151.809 (66.0%)		78.151 (34.0%)	
		80.7%	12.3%	27.3%	53.9%		88.2%	9.9%	26.8%	56.6%

Table 6: Runtime distribution for 10 time steps of the solid body rotation benchmark (cf. Sec. 5.2) without initialization tasks (i.e., only time stepping loop) measured using MATLAB’s profiler. We compare different mesh sizes and approximation orders with runtimes given in seconds. Percentages for assembly and solvers are relative to the runtime of the time stepping loop, percentages for the assembly of **G** and **S** (cf. secs. 4.3.2 and 4.3.1) are relative to assembly runtime, and percentages for inversion of \mathbf{L} (cf. sec. 3.4) and solving (32) for λ are relative to solver step runtimes. Details on the employed hardware and software are given in Table 5 (right).

solid body rotation benchmark presented in Sec. 5.2. Using MATLAB’s profiler to determine the runtime share of each instruction we see — just as expected — that the linear solvers together with the assembly of the time-dependent block matrices, particularly **G** and **S**, are responsible for the majority of the computation time. Table 6 shows the runtime shares for the most expensive parts of the code.

First of all, the overall runtime clearly increases with mesh size and polynomial approximation order simply due to the increasing number of degrees of freedom. More intriguing is the fact that the assembly step becomes more dominant than the linear system solves with the increasing polynomial degree. The primary reason for this is the assembly of matrices \mathbf{G}^m , which do not only grow in size due to the increasing number of degrees of freedom but also require a loop over all quadrature points (see Sec. 4.3.1 for details). With the order of the quadrature rule (and thus the number of quadrature points) increasing with the polynomial degree, the computational complexity of this operation grows quickly. Nevertheless, the total runtime increase for higher polynomial approximation orders is not as pronounced as for the unhybridized DG solver as shown in Sec. 5.2.

In contrast to the shift in the assembly step, the runtime distribution between the block-wise inversion of \mathbf{L} and solving (32) is very similar throughout all approximation orders and different mesh sizes. We would like to point out that the local solves with \mathbf{L}^{-1} are element-local and thus could be easily parallelized with virtually perfect scaling.

6. Register of routines

In this section, the routines added since the first two papers in series [34, 35] are presented in form of a two-part list: first, the scripts implementing the solution algorithm in the order they are executed followed by the alphabetically ordered list of assembly and integration routines used in the solution algorithm. In the code available on GitHub [37], all routines check for correctly provided function arguments using MATLAB/GNU Octave’s `validateattributes` excluded here for brevity.

All data that is needed throughout the entire algorithm (e. g., mesh data structures, pre-computed reference blocks, etc.) are passed between steps in a struct always called `problemData`. In all routines, the argument `g` is a struct containing information about the triangulation. Input parameter `N` is the number of degrees of freedom for the 2D polynomials φ_h , and `Nmu` is the number of degrees of freedom for the 1D polynomials μ_h . Parameter `q0rd` is the order of

the quadrature rule, and basesOnQuad is a struct that contains basis functions φ_h and μ_h evaluated in quadrature points of the reference element \hat{T} , edges \hat{E}_n of the reference element, and reference interval $[0, 1]$. In the time stepping routines, nStep always refers to the current time step index n and nSubStep to the current Runge-Kutta stage i (cf. Sec. 3.3).

6.1. Solution algorithm

problemData = configureProblem(problemData) is executed first and fills the problemData-struct with all basic configuration options. Problem parameters are to be modified inside this routine.

```
function problemData = configureProblem(problemData)
%% Parameters.
% Choose testcase
problemData = setdefault(problemData, 'testcase', 'solid_body');
% Mark run as convergence test (enforces Friedrichs-Keller triangulation)
problemData = setdefault(problemData, 'isConvergence', false);
% Maximum edge length of triangle
problemData = setdefault(problemData, 'hmax', 2^-6);
% Local polynomial approximation order (0 to 4)
problemData = setdefault(problemData, 'p', 2);
% Order of Runge-Kutta method
problemData = setdefault(problemData, 'ordRK', min(problemData.p + 1, 4));
% Order of quadrature rule
problemData = setdefault(problemData, 'qOrd', 2*problemData.p + 1);
% Load testcase
problemData = getTestcase(problemData, problemData.testcase);
% Visualization settings
problemData = setdefault(problemData, 'isVisGrid', false); % visualization of grid
problemData = setdefault(problemData, 'isVisSol', true); % visualization of solution
problemData = setdefault(problemData, 'outputFrequency', 50); % no visualization of every timestep
problemData = setdefault(problemData, 'outputBaseline', ['output' filesep 'hdg_advection']);
problemData = setdefault(problemData, 'outputTypes', {'vtk', 'tec'}); % Type of visualization files
% HDG stabilization parameter
problemData = setdefault(problemData, 'stab', 1.0);
% Enable blockwise local solves and specify block size
problemData = setdefault(problemData, 'isBlockSolve', true);
problemData = setdefault(problemData, 'blockSolveSize', 2^(5 - problemData.p));
%% Domain and triangulation configuration.
% Triangulate unit square using pdeTool (if available or Friedrichs-Keller otherwise).
if ~problemData.isConvergence && license('checkout','PDE_Toolbox')
    problemData.generateGridData = @(hmax) domainPolygon([0 1 1 0], [0 0 1 1], hmax);
else
    fprintf('PDE_Toolbox not available. Using Friedrichs-Keller triangulation.\n');
    problemData.generateGridData = @domainSquare;
end % if
end % function
```

problemData = preprocessProblem(problemData) performs all pre-processing tasks and one-time computations prior to the projection of the initial data and the time stepping loop. This includes mesh generation, evaluation of basis functions in quadrature points, computation of blocks for the reference elements and edges, and assembly of time-independent block matrices.

```
function problemData = preprocessProblem(problemData)
%% Triangulation.
problemData.g = problemData.generateGridData(problemData.hmax);
if problemData.isVisGrid, visualizeGrid(problemData.g); end
%% Globally constant parameters.
problemData.K = problemData.g.numT; % number of triangles
problemData.N = nchoosek(problemData.p + 2, problemData.p); % number of local DOFs
problemData.Nmu = problemData.p + 1; % number of local DOFs on Faces
problemData.dt = problemData.tEnd / problemData.numSteps;
% [K x 3] arrays that mark local edges (EOT) or vertices (VOT) that are
% interior or have a certain boundary type.
problemData.g.markEOTint = problemData.generateMarkEOTint(problemData.g);
problemData.g.markEOTbdr = problemData.generateMarkEOTbdr(problemData.g);
% Precompute some repeatedly evaluated fields
problemData.g = computeDerivedGridData(problemData.g);
%% Configuration output.
fprintf('Computing with polynomial order %d (%d local DOFs) on %d triangles and %d edges.\n', ...
        problemData.p, problemData.N, problemData.K, problemData.g.numE)
qOrd = problemData.qOrd;
N = problemData.N;
```

```

Nmu = problemData.Nmu;
%% Lookup table for basis function.
problemData.basesOnQuad = computeBasesOnQuad(N, struct, [qOrd, qOrd + 1]);
problemData.basesOnQuad = computeBasesOnQuadEdge(Nmu, problemData.basesOnQuad, [qOrd, qOrd + 1]);
%% Computation of matrices on the reference triangle.
problemData.hatM = integrateRefElemPhiPhi(N, problemData.basesOnQuad, qOrd);
hatMmu = integrateRefEdgeMuMu(Nmu, problemData.basesOnQuad, qOrd);
problemData.hatRmu = integrateRefEdgePhiIntMu([N, Nmu], problemData.basesOnQuad, qOrd);
hatRphi = integrateRefEdgePhiIntPhiInt(N, problemData.basesOnQuad, qOrd);
problemData.hatG = integrateRefElemDphiPhiPerQuad(N, problemData.basesOnQuad, qOrd);
problemData.hatS = integrateRefEdgePhiIntMuPerQuad([N, Nmu], problemData.basesOnQuad, qOrd);
%% Assembly of time-independent global matrices.
problemData.globMphi = assembleMatElemPhiPhi(problemData.g, problemData.hatM);
problemData.globRmu = assembleMatEdgePhiIntMu(problemData.g, problemData.g.markE0Tint, ...
problemData.hatRmu);
problemData.globRphi = assembleMatEdgePhiIntPhiInt(problemData.g, problemData.g.markE0Tint, hatRphi);
problemData.globMmuBar = assembleMatEdgeMuMu(problemData.g, problemData.g.markE0Tint, hatMmu);
problemData.globMmuTilde = assembleMatEdgeMuMu(problemData.g, problemData.g.markE0Tbdr, hatMmu);
problemData.globP = problemData.stab .* problemData.globMmuBar + problemData.globMmuTilde;
problemData.globT = problemData.globRmu';
end % function

```

`problemData = initializeProblem(problemData)` projects initial data c^0 and visualizes the initial state.

```

function problemData = initializeProblem(problemData)
problemData.isFinished = false;
%% Initial data on elements and faces
if ~problemData.isStationary
    problemData.cDisc = projectFuncCont2DataDisc(problemData.g, problemData.c0Cont, problemData.qOrd, ...
    problemData.hatM, problemData.basesOnQuad);
    fprintf('L2 error w.r.t. the initial condition: %g\n', computeL2Error(problemData.g, ...
    problemData.cDisc, problemData.c0Cont, problemData.qOrd + 1, problemData.basesOnQuad));
    %% visualization of initial condition.
    if problemData.isVisSol
        cLgr = projectDataDisc2DataLagr(problemData.cDisc);
        visualizeDataLagr(problemData.g, cLgr, 'u_h', problemData.outputBasename, 0, problemData.outputTypes)
    end
    fprintf('Starting time integration from 0 to %g using time step size %g (%d steps).\n', ...
    problemData.tEnd, problemData.dt, problemData.numSteps)
end % if
end % function

```

`problemData = preprocessStep(problemData, nStep)` is the first step in each iteration of the time stepping loop and applies mass matrix \mathbf{M}_φ to solution vector \mathbf{C}^n required in each stage of the DIRK schemes.

```

function problemData = preprocessStep(problemData, nStep)
if ~problemData.isStationary
    problemData.globMcDisc = problemData.globMphi * reshape(problemData.cDisc', [], 1);
end % if
end % function

```

`problemData = solveStep(problemData, nStep)` is the main routine of the time stepping loop and determines Runge-Kutta coefficients before initiating sub-stepping to carry out the Runge-Kutta stages.

```

function problemData = solveStep(problemData, nStep)
if problemData.isStationary
    problemData.t = 0;
else
    % Obtain Runge-Kutta rule and initialize solution vectors
    [problemData.t, problemData.A, problemData.b] = rungeKuttaImplicit(problemData.ordRK, ...
    problemData.dt, (nStep - 1) * problemData.dt);
    problemData.cDiscRK = cell(length(problemData.t), 1);
end % if
% Carry out RK steps
problemData.isSubSteppingFinished = false;
problemData = iterateSubSteps(problemData, nStep);
end % function

```

`problemData = preprocessSubStep(problemData, nStep, nSubStep)` is the main assembly routine of the current Runge-Kutta stage, where velocity function \mathbf{u} is evaluated at quadrature points of all edges, inflow and outflow boundaries are determined, and all time-dependent block matrices and right-hand-side vectors are built.

```

function problemData = preprocessSubStep(problemData, nStep, nSubStep)
% Select time level for current Runge-Kutta stage
t = problemData.t(nSubStep);
cDCont = @(x1, x2) problemData.cDCont(t, x1, x2);
u1Cont = @(x1, x2) problemData.u1Cont(t, x1, x2);
u2Cont = @(x1, x2) problemData.u2Cont(t, x1, x2);
fCont = @(x1,x2) problemData.fCont(t, x1, x2);
% Evaluate normal advection velocity on every edge.
uNormalQOEOT = computeFuncContNuOnQuadEdge(problemData.g, u1Cont, u2Cont, problemData.qOrd);
% Determine inflow- and outflow edges
[~, W] = quadRule1D(problemData.qOrd);
markEOTbdrOut = problemData.g.markEOTbdr & sum(bsxfun(@times, reshape(W,1,1,[],), uNormalQOEOT), 3) > 0;
markEOTbdrIn = problemData.g.markEOTbdr & ~markEOTbdrOut;
% Assemble source term.
problemData.globH = problemData.globMphi * reshape(projectFuncCont2DataDisc(problemData.g, fCont, ...
    problemData.qOrd, problemData.hatM, problemData.basesOnQuad)', [], 1);
% Assemble element integral contributions.
problemData.globG = assembleMatElemDphiPhiFuncContVec(problemData.g, problemData.hatG, ...
    u1Cont, u2Cont, problemData.qOrd);
% Assemble flux on interior and outflow edges
problemData.globS = assembleMatEdgePhiIntMuVal(problemData.g, problemData.g.markEOTint|markEOTbdrOut,...
    problemData.hatS, uNormalQOEOT);
problemData.globKmuOut = assembleMatEdgePhiIntMu(problemData.g, markEOTbdrOut, problemData.hatRmu)';
% Assemble inflow boundary conditions.
cQOEOT = computeFuncContOnQuadEdge(problemData.g, cDCont, problemData.qOrd);
problemData.globFphiIn = assembleVecEdgePhiIntVal(problemData.g, markEOTbdrIn, ...
    cQOEOT .* uNormalQOEOT, problemData.N, problemData.basesOnQuad);
problemData.globKmuIn = assembleVecEdgeMuFuncCont(problemData.g, markEOTbdrIn, ...
    cDCont, problemData.basesOnQuad, problemData.qOrd);
end % function

```

problemData = solveSubStep(problemData, nStep, nSubStep) builds and solves the linear systems described in Sec. 3.4 for the current Runge-Kutta stage.

```

function problemData = solveSubStep(problemData, nStep, nSubStep)
K = problemData.K;
N = problemData.N;
if problemData.isStationary
    matL = - problemData.globG{1} - problemData.globG{2} + problemData.stab * problemData.globRphi;
    vecQ = problemData.globH - problemData.globFphiIn;
    matM = problemData.globS - problemData.stab * problemData.globRmu;
else
    dtA = problemData.dt * problemData.A;
    cDiscRkRHS = zeros(K * N, 1);
    for i = 1 : nSubStep - 1
        cDiscRkRHS = cDiscRkRHS + dtA(nSubStep, i) .* problemData.cDiscRK{i};
    end
    matLbar = - problemData.globG{1} - problemData.globG{2} + problemData.stab * problemData.globRphi;
    matL = problemData.globMphi + dtA(nSubStep, nSubStep) .* matLbar;
    vecBphi = problemData.globH - problemData.globFphiIn;
    vecQ = problemData.globMcDisc + dtA(nSubStep, nSubStep) * vecBphi + cDiscRkRHS;
    matMbar = problemData.globS - problemData.stab * problemData.globRmu;
    matM = dtA(nSubStep, nSubStep) .* matMbar;
end % if
%% Computing local solves
if problemData.isBlockSolve
    matLinv = blkinv(matL, problemData.blockSolveSize * N);
    LinvQ = matLinv * vecQ;
    LinvM = matLinv * matM;
else
    LinvQ = matL \ vecQ;
    LinvM = matL \ matM;
end % if
%% Solving global system for lambda
matN = - problemData.stab * problemData.globT - problemData.globKmuOut ;
matP = problemData.globP;
lambdaDisc = (-matN * LinvM + matP) \ (problemData.globKmuIn - matN * LinvQ);
%% Reconstructing local solutions from updated lambda
problemData.cDisc = LinvQ - LinvM * lambdaDisc;
if ~problemData.isStationary
    problemData.cDiscRK{nSubStep} = vecBphi - matLbar * problemData.cDisc - matMbar * lambdaDisc;
end % if
problemData.cDisc = reshape(problemData.cDisc, N, K)';
end % function

```


`problemData = postprocessSubStep(problemData, nStep, nSubStep)` determines whether ...SubStep-routines must be executed once more to carry out the remaining Runge-Kutta stages or if control has to be returned to function `solveStep`.

```
function problemData = postprocessSubStep(problemData, nStep, nSubStep)
problemData.isSubSteppingFinished = problemData.isStationary || nSubStep >= length(problemData.t);
end % function
```

`problemData = postprocessStep(problemData, nStep)` checks if all time integration steps have been carried out or if another iteration of the time stepping loop is required.

```
function problemData = postprocessStep(problemData, nStep)
problemData.isFinished = problemData.isStationary || nStep >= problemData.numSteps;
end % function
```

`problemData = outputStep(problemData, outputStep)` writes visualization output files for the updated solution.

```
function problemData = outputStep(problemData, nStep)
%% visualization
if problemData.isVisSol && mod(nStep, problemData.outputFrequency) == 0
    cLagrange = projectDataDisc2DataLagr(problemData.cDisc);
    visualizeDataLagr(problemData.g, cLagrange, 'u_h', problemData.outputBasename, ...
        nStep, problemData.outputTypes);
end % if
end % function
```

`problemData = postprocessProblem(problemData)` is executed after termination of the time stepping loop and computes the L^2 -error of the solution if the analytical solution is provided.

```
function problemData = postprocessProblem(problemData)
%% Visualization
if problemData.isVisSol
    cLagrange = projectDataDisc2DataLagr(problemData.cDisc);
    visualizeDataLagr(problemData.g, cLagrange, 'u_h', problemData.outputBasename, ...
        problemData.numSteps, problemData.outputTypes);
end % if
fprintf('Finished simulation at t_end=%g\n', problemData.tEnd);
%% Error evaluation
if isfield(problemData, 'cCont')
    problemData.error = computeL2Error(problemData.g, problemData.cDisc, ...
        @(x1, x2) problemData.cCont(problemData.tEnd, x1, x2), problemData.qOrd, problemData.basesOnQuad);
    fprintf('L2 error w.r.t. the analytical solution: %g\n', problemData.error);
else
    problemData.error = computeL2Error(problemData.g, problemData.cDisc, ...
        problemData.c0Cont, problemData.qOrd, problemData.basesOnQuad);
    fprintf('L2 error w.r.t. the initial condition: %g\n', problemData.error);
end % if
end % function
```

6.2. Helper routines

`ret = assembleMatEdgeMuMu(g, markEOT, refEdgeMuMu)` assembles mass-matrices $\bar{\mathbf{M}}_\mu$ and $\tilde{\mathbf{M}}_\mu$ for the edge-based basis functions μ_h as described in Sec. 4.3.4. Array `markEOT` plays the role of the Kronecker delta in the matrix definition by selecting the relevant edges for either matrix. `refEdgeMuMu` is the mass matrix $\hat{\mathbf{M}}_\mu$ on the reference interval $[0, 1]$ computed by `integrateRefEdgeMuMu`.

```
function ret = assembleMatEdgeMuMu(g, markEOT, refEdgeMuMu)
Nmu = size(refEdgeMuMu, 1); Kedge = g.numE;
ret = sparse(Kedge * Nmu, Kedge * Nmu);
for n = 1 : 3
    Kkn = g.areaEOT(:, n) .* markEOT(:, n);
    ret = ret + kron(sparse(g.EOT(:, n), g.EOT(:, n), Kkn, Kedge, Kedge), refEdgeMuMu);
end % for
end % function
```

`ret = assembleMatEdgePhiIntMu(g, markEOT, refEdgePhiIntMu)` assembles matrices \mathbf{R}_μ , \mathbf{T} , and $\mathbf{K}_{\mu,\text{out}}$ as detailed in Sec. 4.3.3. As before, `markEOT` is used to select relevant edges, and `refEdgePhiIntMu` stores the pre-computed reference blocks $\hat{\mathbf{R}}_\mu$ given by `integrateRefEdgePhiIntMu`.

```
function ret = assembleMatEdgePhiIntMu(g, markEOT, refEdgePhiIntMu)
K = g.numT; Kedge = g.numE;
[N, Nmu, ~, ~] = size(refEdgePhiIntMu);
ret = sparse(K * N, Kedge * Nmu);
for n = 1 : 3
    for l = 1 : 2
        Rkn = markEOT(:,n) .* g.markSideEOT(:, n, l) .* g.areaEOT(:, n);
        ret = ret + kron(sparse(1 : K, g.EOT(:, n), Rkn, K, Kedge), refEdgePhiIntMu(:, :, n, l));
    end % for
end % for
end % function
```

`ret = assembleMatEdgePhiIntMuVal(g, markEOT, refEdgePhiIntMuPerQuad, valOnQuad)` builds the global matrices \mathbf{S} and \mathbf{S}_{out} given in Sec. 4.3.2. Both matrices are assembled at once by specifying all interior and outflow edges as `markEOT`. The reference blocks per quadrature point $\hat{\mathbf{S}}$ are computed by `integrateRefEdgePhiIntMuPerQuad` and specified in the parameter `refEdgePhiIntMuPerQuad`. Parameter `valOnQuad` stores the normal velocity \mathbf{U}_ν evaluated at the quadrature points of each edge.

```
function ret = assembleMatEdgePhiIntMuVal(g, markEOT, refEdgePhiIntMuPerQuad, valOnQuad)
K = g.numT; Kbar = g.numE;
[N, Nmu, ~, R] = size(refEdgePhiIntMuPerQuad{1});
ret = sparse(K*N, Kbar*Nmu);
for n = 1 : 3
    RknTimesVal = sparse(K*N, Nmu);
    markAreaEOT = markEOT(:, n) .* g.areaEOT(:, n);
    for l = 1 : 2
        markAreaSideEOT = markAreaEOT .* g.markSideEOT(:, n, l);
        for r = 1 : R
            RknTimesVal = RknTimesVal + kron(markAreaSideEOT .* valOnQuad(:, n, r), ...
                refEdgePhiIntMuPerQuad{1}(:, :, n, r));
        end % for r
    end % for l
    ret = ret + kronVec(sparse(1:K, g.EOT(:, n), ones(K, 1), K, Kbar), RknTimesVal);
end % for n
end % function
```

`ret = assembleMatElemDphiPhiFuncContVec(g, refElemDphiPhiPerQuad, funcCont1, funcCont2, qOrd)` assembles matrices \mathbf{G}^m described in Sec. 4.3.1. Reference blocks $\hat{\mathbf{G}}$ are provided in parameter `refElemDphiPhiPerQuad` and computed by `integrateRefElemDphiPhiPerQuad`. The two components of the continuous function in the integrand – here the velocity components – are given as function handles in `funcCont1` and `funcCont2`, respectively.

```
function ret = assembleMatElemDphiPhiFuncContVec(g, refElemDphiPhiPerQuad, funcCont1, funcCont2, qOrd)
K = g.numT; [N, ~, R] = size(refElemDphiPhiPerQuad{1});
if nargin < 5, p = (sqrt(8*N+1)-3)/2; qOrd = max(2*p, 1); end
[Q1, Q2, ~] = quadRule2D(qOrd);
ret = { zeros(K*N, N), zeros(K*N,N) };
for r = 1 : R
    valOnQuad1 = funcCont1(g.mapRef2Phy(1, Q1(r), Q2(r)), g.mapRef2Phy(2, Q1(r), Q2(r)));
    ret{1} = ret{1} + kron(g.B(:,2,2) .* valOnQuad1, refElemDphiPhiPerQuad{1}(:, :, r)) ...
        - kron(g.B(:,2,1) .* valOnQuad1, refElemDphiPhiPerQuad{2}(:, :, r));
    valOnQuad2 = funcCont2(g.mapRef2Phy(1, Q1(r), Q2(r)), g.mapRef2Phy(2, Q1(r), Q2(r)));
    ret{2} = ret{2} - kron(g.B(:,1,2) .* valOnQuad2, refElemDphiPhiPerQuad{1}(:, :, r)) ...
        + kron(g.B(:,1,1) .* valOnQuad2, refElemDphiPhiPerQuad{2}(:, :, r));
end % for
ret{1} = kronVec(speye(K,K), ret{1});
ret{2} = kronVec(speye(K,K), ret{2});
end % function
```

`ret = assembleVecEdgeMuFuncCont(g, markEOT, funcCont, basesOnQuad, qOrd)` builds the vector with Dirichlet data $\mathbf{K}_{\mu,\text{in}}$ (cf. Sec. 4.3.5). Inflow edges are specified in `markEOT`, and Dirichlet data is specified as a function handle in `funcCont`.

```

function ret = assembleVecEdgeMuFuncCont(g, markEOT, funcCont, basesOnQuad, qOrd)
[Q, W] = quadRule1D(qOrd);
[R, Nmu] = size(basesOnQuad.mu{qOrd});
ret = zeros(g.numE, Nmu);
for n = 1 : 3
    [Q1, Q2] = gammaMap(n, Q);
    funcOnQuad = funcCont(g.mapRef2Phy(1, Q1, Q2), g.mapRef2Phy(2, Q1, Q2));
    Kkn = markEOT(:, n) .* g.areaEOT(:, n);
    ret(g.EOT(:, n), :) = ret(g.EOT(:, n), :) + (repmat(Kkn, 1, R) .* funcOnQuad) ...
        * ( repmat(W', 1, Nmu) .* basesOnQuad.mu{qOrd} );
end % for
ret = reshape(ret', [], 1);
end % function

```

`invA = blkinv(A, blockSize)` inverts a square block-diagonal matrix A built up from blocks of size $n \times n$, which is used to compute L^{-1} as described in Sec. 3.4. To improve the computational performance of this method, the inversion is performed using a blocking-technique where a pre-specified number of diagonal blocks are inverted together. Parameter `blockSize` specifies the blocking size and must be a multiple of n .

```

function invA = blkinv(A, blockSize)
n = size(A,1);
numBlocks = ceil(n / blockSize);
idxEnd = (numBlocks - 1) * blockSize;
idxs = mat2cell(1 : idxEnd, 1, ones(1, numBlocks - 1) * blockSize);
cellA = cellfun(@(X) A(X, X), idxs, 'UniformOutput', false);
cellInvA = cellfun(@(X) inv(X), cellA, 'UniformOutput', false);
invA = blkdiag(cellInvA{:}, inv(A(idxEnd + 1 : end, idxEnd + 1 : end)));
end % function

```

`basesOnQuadEdge = computeBasesOnQuadEdge(N, basesOnQuadEdge, requiredOrders)` evaluates edge basis function μ_h at the quadrature points of the reference interval $[0, 1]$. Parameter `basesOnQuadEdge` is a (possibly empty) struct to which the computed fields `mu` and `thetaMu` (the latter corresponding to $\hat{\mu} \circ \hat{\beta}_{kn}$) are added. The quadrature orders can be optionally specified in `requiredOrders` and with defaults equal to $2p$ and $2p + 1$.

```

function basesOnQuadEdge = computeBasesOnQuadEdge(N, basesOnQuadEdge, requiredOrders)
if nargin < 3
    p = N - 1;
    if p > 0
        requiredOrders = [2*p, 2*p+1];
    else
        requiredOrders = 1;
    end % if
end % if
basesOnQuadEdge.mu = cell(max(requiredOrders), 1);
basesOnQuadEdge.thetaMu = cell(max(requiredOrders), 1);
for it = 1 : length(requiredOrders)
    ord = requiredOrders(it);
    [Q, ~] = quadRule1D(ord);
    R = length(Q);
    basesOnQuadEdge.mu{ord} = zeros(R, N);
    for i = 1 : N
        basesOnQuadEdge.mu{ord}(:, i) = phi1D(i, Q);
    end
    basesOnQuadEdge.thetaMu{ord} = zeros(R, N, 2);
    basesOnQuadEdge.thetaMu{ord}(:, :, 1) = basesOnQuadEdge.mu{ord};
    basesOnQuadEdge.thetaMu{ord}(:, :, 2) = flipud(basesOnQuadEdge.mu{ord});
end % for
end % function

```

`g = computeDerivedGridData(g)` enriches grid data structure `g` by an additional $K \times 3 \times 2$ field `markSideEOT` that marks for each element-local edge, whether the element has local index 1 or 2 at the edge thus providing a way to determine l in the mapping κ (cf. eq. (6)) for given element T_k and edge $E_{kn} = E_{\bar{k}}$.

```

function g = computeDerivedGridData(g)
g.markSideEOT = true(g.numT, 3, 2);
for n = 1 : 3
    % Mark element to have local id 1 or 2 at the given edge
    g.markSideEOT(:, n, 1) = g.TOE(g.EOT(:, n), 2) ~= (1:g.numT)';
end

```

```

g.markSideEOT(:, n, 2) = ~g.markSideEOT(:, n, 1);
end % for
end % function

```

`ret = computeFuncContOnQuadEdge(g, funcCont, qOrd)` evaluates a given function handle `funcCont` in all quadrature points of all edges. This is used to determine the Dirichlet boundary data c_D before multiplying it with the normal velocity and assembling $\mathbf{F}_{\varphi, \text{in}}$ in routine `assembleVecEdgePhiIntVal`.

```

function ret = computeFuncContOnQuadEdge(g, funcCont, qOrd)
K = g.numT; [Q, W] = quadRule1D(qOrd);
ret = zeros(K, 3, length(W));
for n = 1 : 3
    [Q1, Q2] = gammaMap(n, Q);
    ret(:, n, :) = funcCont(g.mapRef2Phy(1, Q1, Q2), g.mapRef2Phy(2, Q1, Q2));
end % for
end % function

```

`problemData = getTestcase(problemData, testcase)` defines initial and boundary data as well as the analytical solution, if available, for test cases given in Sec. 5. To shorten the presentation, the actual definitions are omitted here but can be found in GitHub [37].

```

function problemData = getTestcase(problemData, testcase)
switch testcase
case 'solid_body'
    % LeVeque's solid body rotation
case 'stationary'
    % Stationary analytical example
case 'transient_ode'
    % Transient ODE example
case 'transient'
    % Transient analytical example
otherwise
    error('Invalid testcase "%s".', testcase);
end % switch
fprintf('Loaded testcase "%s".\n', testcase);
% Time stepping parameters
problemData = setdefault(problemData, 'numSteps', numSteps);
problemData = setdefault(problemData, 'tEnd', tEnd);
problemData = setdefault(problemData, 'isStationary', isStationary);
% Store defined functions in struct
problemData = setdefault(problemData, 'c0Cont', c0Cont);
problemData = setdefault(problemData, 'fCont', fCont);
problemData = setdefault(problemData, 'u1Cont', u1Cont);
problemData = setdefault(problemData, 'u2Cont', u2Cont);
problemData = setdefault(problemData, 'cDCont', cDCont);
if isAnalytical, problemData = setdefault(problemData, 'cCont', cCont); end
% Specify boundary conditions
problemData = setdefault(problemData, 'generateMarkEOTint', @(g) g.idEOT == 0);
problemData = setdefault(problemData, 'generateMarkEOTbdr', @(g) g.idEOT ~= 0);
end % function

```

`ret = integrateRefEdgeMuMu(N, basesOnQuadEdge, qOrd)` computes reference blocks $\hat{\mathbf{M}}_\mu$ (see Sec. 4.3.4).

```

function ret = integrateRefEdgeMuMu(N, basesOnQuad, qOrd)
if nargin < 3, p = N-1; qOrd = 2*p+1; end
[~, W] = quadRule1D(qOrd);
ret = zeros(N, N); % [N x N]
for i = 1 : N
    for j = 1 : N
        ret(i, j) = sum( W' .* basesOnQuad.mu{qOrd}(:, i) .* basesOnQuad.mu{qOrd}(:, j) );
    end % for
end % for
end % function

```

`ret = integrateRefEdgePhiIntMu(N, basesOnQuad, qOrd)` computes reference blocks $\hat{\mathbf{R}}_\mu$ (see Sec. 4.3.3).

```

function ret = integrateRefEdgePhiIntMu(N, basesOnQuad, qOrd)
if nargin < 3, p = (sqrt(8*N(1)+1)-3)/2; qOrd = 2*p+1; end
[~, W] = quadRule1D(qOrd);
ret = zeros(N(1), N(2), 3, 2); % [N(1) x N(2) x 3 x 2]
for n = 1 : 3

```

```

    for i = 1 : N(1)
        for j = 1 : N(2)
            ret(i, j, n, 1) = W * (basesOnQuad.phi1D{q0rd}(:, i, n) .* basesOnQuad.mu{q0rd}(:, j) );
            ret(i, j, n, 2) = W * (basesOnQuad.phi1D{q0rd}(:, i, n) .* basesOnQuad.thetaMu{q0rd}(:, j,
                ↪2) );
        end % for
    end % for
end % for
end % function

```

`ret = integrateRefEdgePhiIntMuPerQuad(N, basesOnQuad, q0rd)` computes reference blocks $\hat{\mathbf{S}}$ (see Sec. 4.3.2).

```

function ret = integrateRefEdgePhiIntMuPerQuad(N, basesOnQuad, q0rd)
if nargin < 3, p = (sqrt(8*N+1)-3)/2; q0rd = 2*p+1; end
[, W] = quadRule1D(q0rd); R = length(W);
ret = { zeros(N(1), N(2), 3, R); zeros(N(1), N(2), 3, R) };
for n = 1 : 3
    for i = 1 : N(1)
        for j = 1 : N(2)
            ret{1}(i, j, n, :) = W(:) .* basesOnQuad.phi1D{q0rd}(:, i, n) ...
                .* basesOnQuad.mu{q0rd}(:, j);
            ret{2}(i, j, n, :) = W(:) .* basesOnQuad.phi1D{q0rd}(:, i, n) ...
                .* basesOnQuad.thetaMu{q0rd}(:, j, 2);
        end % for j
    end % for i
end % for n
end % function

```

`ret = integrateRefElemDphiPhiPerQuad(N, basesOnQuad, q0rd)` computes reference blocks $\hat{\mathbf{G}}$ (see Sec. 4.3.1).

```

function ret = integrateRefElemDphiPhiPerQuad(N, basesOnQuad, q0rd)
if nargin < 3, p = (sqrt(8*N+1)-3)/2; q0rd = max(2*p, 1); end
[, ~, W] = quadRule2D(q0rd); R = length(W);
ret = { zeros(N, N, R); zeros(N, N, R) };
for i = 1 : N
    for j = 1 : N
        for m = 1 : 2
            ret{m}(i, j, :) = W(:) .* basesOnQuad.phi2D{q0rd}(:, j) .* basesOnQuad.gradPhi2D{q0rd}(:, i, m);
        end % for
    end % for j
end % for i
end % function

```

`[t, A, b, c] = rungeKuttaImplicit(ord, tau, t0)` provides the Butcher tableau for DIRK schemes of orders one to four (see Sec. 3.3) with the order given in `ord`. The current time level is provided in `t0`, and the time step size is given in `tau`.

```

function [t, A, b, c] = rungeKuttaImplicit(ord, tau, t0)
switch ord
case 1
    A = 1;
case 2
    lambda = 0.5 * (2 - sqrt(2));
    A = [ lambda, 0;
          (1-lambda), lambda ];
case 3
    alph = 0.4358665215084589994160194511935568425292;
    bet = 0.5 * (1.+ alph);
    b1 = -0.25 * (6 * alph * alph - 16 * alph + 1);
    b2 = 0.25 * (6 * alph * alph - 20 * alph + 5);
    A = [ alph, 0, 0;
          bet - alph, alph, 0;
          b1, b2, alph ];
case 4
    gamm = 1./4.;
    A = [ gamm, 0, 0, 0, 0;
          1./2., gamm, 0, 0, 0;
          17./50., -1./25., gamm, 0, 0;
          371./1360., -137./2720., 15./544., gamm, 0;
          25./24., -49./48., 125./16., -85./12., gamm ];
otherwise
    error('Invalid order for implicit Runge-Kutta')
end % switch

```

```
b = A(end, :);  
c = sum(A, 2);  
t = t0 + c * tau;  
end % function
```

7. Conclusion and Outlook

The third work in our series introduces a hybridized DG formulation and expands our FESTUNG framework to include edge-based degrees of freedom as well as implicit time stepping schemes. A comparison of the results obtained using HDG to those from the standard DG method indicates little difference in solution quality (except in the lowest order approximation $p = 0$ case, where the unhybridized DG held a slight edge) and confirms the well-known computational cost advantage of the HDG method for higher order polynomial spaces. Since HDG implementations spend a significant portion of computing time in element-local solves, this advantage over standard DG discretizations becomes even more pronounced for parallel implementations using the distributed memory programming paradigm. Our future work plans include more complex systems of PDEs and coupled multi-physics applications.

Acknowledgments

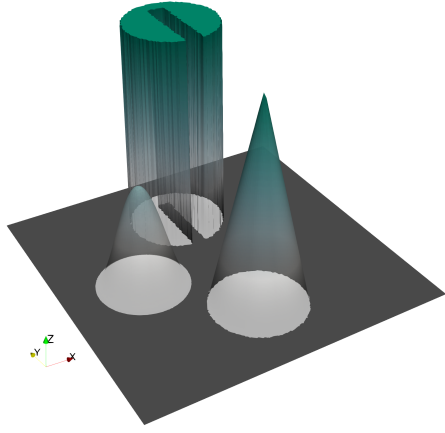
The stay of A. Jaust at the University of Erlangen-Nürnberg was supported by the Research Foundation - Flanders (FWO) with a grant for a short study visit abroad and by the Special Research Fund (BOF) of Hasselt University.

Index of notation

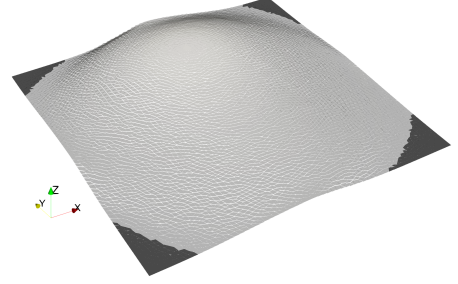
Symbol	Definition
$\text{diag}(\mathbf{A}, \mathbf{B})$	$:= \begin{bmatrix} \mathbf{A} & \\ & \mathbf{B} \end{bmatrix}$, block-diagonal matrix with blocks \mathbf{A}, \mathbf{B} .
$\mathbf{a} \cdot \mathbf{b}$	$:= \sum_{m=1}^2 a_m b_m$, Euclidean scalar product in \mathbb{R}^2 .
∇	$:= [\partial_{x^1}, \partial_{x^2}]^T$, spatial gradient in the physical domain Ω .
\circ	Composition of functions or Hadamard product.
\otimes, \otimes_V	Kronecker product, ‘vectorial’ Kronecker product.
α	Stabilization parameter.
$\hat{\beta}_{kn}$	Mapping from $[0, 1]$ to $[0, 1]$ that adapts the edge orientation.
c	Concentration (scalar-valued unknown).
$c_{\partial\Omega}$	Concentration on the boundary $\partial\Omega$.
c^0, c_D	Concentration prescribed at initial time $t = 0$ and on the inflow Dirichlet boundary.
\mathbf{C}	$\in \mathbb{R}^{KN}$, representation vector of $c_h \in \mathbb{P}_p(\mathcal{T}_h)$ with respect to $\{\varphi_{kj}\}$.
$\delta_{[\text{condition}]}$	$:= \{1 \text{ if condition is true, } 0 \text{ otherwise}\}$, Kronecker delta.
E_{kn}, \hat{E}_n	n th edge of the physical triangle T_k , n th edge of the reference triangle \hat{T} .
$E_{\bar{k}}$	\bar{k} th physical edge of the mesh.
\mathcal{E}, \mathcal{T}	Sets of edges and triangles.
$\mathcal{E}_{\text{int}}, \mathcal{E}_{\text{ext}}, \mathcal{E}_{\text{in}}, \mathcal{E}_{\text{out}}$	Sets of interior, boundary, inflow, and outflow edges.
$\mathbf{f}, \hat{\mathbf{f}}$	Advective flux function, numerical flux.
\mathbf{F}_k	Affine mapping from \hat{T} to T_k .
Γ	Trace of the mesh.
$\hat{\gamma}_n$	Mapping from $[0, 1]$ to \hat{E}_n .
h	Mesh fineness of \mathcal{T}_h .
h_T	$:= \text{diam}(T)$, diameter of triangle $T \in \mathcal{T}_h$.
$\xi(t, \mathbf{x})$	Source / sink (scalar-valued coefficient function).
J	$:= (t^0, t_{\text{end}})$, open time interval.
K, \bar{K}	Number of triangles, number of edges.
$\kappa(\bar{k}, l)$	Index mapping from physical edge to neighboring physical triangles T_{k^-}, T_{k^+} .
λ_h	Scalar-valued unknown on the trace of the mesh.
Λ	$\in \mathbb{R}^{KN}$, representation vector of $\lambda_h \in \mathbb{P}_p(\mathcal{E}_h)$ with respect to $\{\mu_{kj}\}$.
$\mu_{\bar{k}i}, \hat{\mu}_i$	i th hierarchical basis function on $E_{\bar{k}}$, i th hierarchical basis function on $[0, 1]$.
$\mathbf{v}_T, \mathbf{v}_{kn}, \mathbf{v}_{\bar{k}}$	Unit normal on ∂T pointing outward of T , unit normal on E_{kn} , unit normal on $E_{\bar{k}}$.
N	$:= (p+1)(p+2)/2$, number of local degrees of freedom of $\mathbb{P}_p(T)$.
\bar{N}	$:= p+1$, number of local degrees of freedom of $\mathbb{P}_p(E)$.
ω_r	Quadrature weight associated with $\hat{\mathbf{q}}_r$.
$\Omega, \partial\Omega, \partial\Omega_{\text{in}}, \partial\Omega_{\text{out}}$	spatial domain in two dimensions, boundary of Ω , inflow- and outflow boundaries, $\partial\Omega = \partial\Omega_{\text{in}} \cup \partial\Omega_{\text{out}}$.
p	$= (\sqrt{8N+1} - 3)/2$, polynomial degree.
$\varphi_{ki}, \hat{\varphi}_i$	i th hierarchical basis function on T_k , i th hierarchical basis function on \hat{T} .
$\mathbb{P}_p(T)$	Space of polynomials on $T \in \mathcal{T}_h$ of degree at most p .
$\mathbb{P}_p(\mathcal{T}_h)$	$:= \{w_h : \bar{\Omega} \rightarrow \mathbb{R} ; \forall T \in \mathcal{T}_h, w_h _T \in \mathbb{P}_p(T)\}$.
$\mathbb{P}_p(\mathcal{E}_h)$	$:= \{w_h : \Gamma \rightarrow \mathbb{R} ; \forall E \in \mathcal{E}_h, w_h _E \in \mathbb{P}_p(E)\}$.
$R, \hat{\mathbf{q}}_r$	Number of quadrature points, r th quadrature point in \hat{T} .
$\rho(k, n)$	Index mapping from edge E_{kn} to physical edge $E_{\bar{k}}$.
t	Time variable.
t^n	n th time level.
t_{end}	End time.
$\hat{\theta}_{n^-n^+}$	Mapping from \hat{E}_{n^-} to \hat{E}_{n^+} .
Δt^n	$:= t^{n+1} - t^n$, time step size.
$T_k, \partial T_k$	k th physical triangle, boundary of T_k .
\hat{T}	Bi-unit reference triangle.
\mathbf{u}	Velocity (vector-valued coefficient function).
\mathbf{x}	$= [x^1, x^2]^T$, space variable in the physical domain Ω .
$\hat{\mathbf{x}}$	$= [\hat{x}^1, \hat{x}^2]^T$, space variable in the reference triangle \hat{T} .

- [1] H. Reed, T. R. Hill, Triangular mesh methods for the neutron transport equation, Tech. Rep. LA-UR-73-479, Los Alamos Scientific Laboratory, NM (1973).
- [2] D. N. Arnold, F. Brezzi, B. Cockburn, L. D. Marini, Unified analysis of discontinuous Galerkin methods for elliptic problems, *SIAM Journal on Numerical Analysis* 39 (2002) 1749–1779.
- [3] C.-W. Shu, Discontinuous Galerkin method for time-dependent problems: survey and recent developments, in: *Recent developments in discontinuous Galerkin finite element methods for partial differential equations*, Vol. 157 of IMA Vol. Math. Appl., Springer, Cham, 2014, pp. 25–62. doi:10.1007/978-3-319-01818-8_2.
- [4] F. Bassi, L. Botti, A. Colombo, D. D. Pietro, P. Tesini, On the flexibility of agglomeration based physical space discontinuous Galerkin discretizations, *Journal of Computational Physics* 231 (2011) 45–65.
- [5] L. Wang, P.-O. Persson, A high-order discontinuous Galerkin method with unstructured spacetime meshes for two-dimensional compressible flows on domains with large deformations, *Computers & Fluids* 118 (2015) 53 – 68. doi:10.1016/j.compfluid.2015.05.026.
- [6] P. Ortwein, T. Binder, S. Copplestone, A. Mirza, P. Nizenkov, M. Pfeiffer, T. Stindl, S. Fasoulas, C.-D. Munz, Parallel performance of a discontinuous Galerkin spectral element method based PIC-DSMC solver, in: W. E. Nagel, D. H. Krner, M. M. Resch (Eds.), *High Performance Computing in Science and Engineering* 14, Springer, 2015, pp. 671–681. doi:10.1007/978-3-319-10810-0_44.
- [7] K. Fidkowski, T. Oliver, J. Lu, D. L. Darmofal, p-Multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier–Stokes equations, *Journal of Computational Physics* 207 (2005) 92–113.
- [8] H. Luo, J. D. Baum, R. Löhner, A p-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids, *Journal of Computational Physics* 211 (2) (2006) 767–783. doi:10.1016/j.jcp.2005.06.019.
URL <http://dx.doi.org/10.1016/j.jcp.2005.06.019>
- [9] F. Bassi, A. Ghidoni, S. Rebay, P. Tesini, High-order accurate p-multigrid discontinuous Galerkin solution of the euler equations, *International Journal for Numerical Methods in Fluids* 60 (8) (2009) 847–865. doi:10.1002/flid.1917.
URL <http://dx.doi.org/10.1002/flid.1917>
- [10] V. Aizinger, D. Kuzmin, L. Korosus, Scale separation in fast hierarchical solvers for discontinuous Galerkin methods, *Applied Mathematics and Computation* 266 (2015) 838–849. doi:10.1016/j.amc.2015.05.047.
URL <http://www.sciencedirect.com/science/article/pii/S0096300315006645>
- [11] A. Jaust, J. Schütz, V. Aizinger, An efficient linear solver for the hybridized discontinuous Galerkin method, *PAMM* 16 (1) (2016) 845–846. doi:10.1002/pamm.201610411.
URL <http://dx.doi.org/10.1002/pamm.201610411>
- [12] J. Schütz, V. Aizinger, A hierarchical scale separation approach for the hybridized discontinuous Galerkin method, *Journal of Computational and Applied Mathematics* 317 (2017) 500–509. doi:10.1016/j.cam.2016.12.018.
URL <http://www.sciencedirect.com/science/article/pii/S0377042716306288>
- [13] C. Thiele, M. Araya-Polo, F. O. Alpak, B. Riviere, F. Frank, Inexact hierarchical scale separation: A two-scale approach for linear systems from discontinuous Galerkin discretizations, *Computers and Mathematics with Applications* (2017) in pressdoi:<http://dx.doi.org/10.1016/j.camwa.2017.06.025>.
URL <http://www.sciencedirect.com/science/article/pii/S0898122117303735>
- [14] R. M. Kirby, S. J. Sherwin, B. Cockburn, To CG or HDG: A comparative study, *Journal of Scientific Computing* 51 (1) (2012) 183–212. doi:10.1007/s10915-011-9501-7.
- [15] S. Yakovlev, D. Moxey, R. M. Kirby, S. J. Sherwin, To CG or to HDG: A comparative study in 3d, *Journal of Scientific Computing* 67 (1) (2016) 192–220. doi:10.1007/s10915-015-0076-6.
- [16] M. Woopen, A. Balan, G. May, J. Schütz, A comparison of hybridized and standard DG methods for target-based *hp*-adaptive simulation of compressible flow, *Computers and Fluids* 98 (2014) 3–16.
- [17] B. F. de Veubeke, Displacement and equilibrium models in the finite element method., in: O.C.Zienkiewicz, G.S.Holister (Eds.), *Stress Analysis*, Wiley New York, 1965, Ch. 9, pp. 145–197.
- [18] D. Arnold, F. Brezzi, Mixed and nonconforming Finite Element methods: Implementation, postprocessing and error estimates, *Mathematical Modelling and Numerical Analysis* 19 (1985) 7–32.
- [19] F. Brezzi, J. Douglas, L. D. Marini, Two families of mixed finite elements for second order elliptic problems, *Numerische Mathematik* 47 (1985) 217–235.
- [20] B. Cockburn, J. Gopalakrishnan, A characterization of hybridized mixed methods for second order elliptic problems, *SIAM Journal on Numerical Analysis* 42 (2004) 283–301.
- [21] B. Cockburn, J. Gopalakrishnan, R. Lazarov, Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems, *SIAM Journal on Numerical Analysis* 47 (2009) 1319–1365.
- [22] B. Cockburn, J. Gopalakrishnan, N. C. Nguyen, J. Peraire, F.-J. Sayas, Analysis of HDG methods for Stokes flow, *Mathematics of Computation* 80 (274) (2011) 723–760. doi:10.1090/S0025-5718-2010-02410-X.
- [23] H. Egger, C. Waluga, *hp*-analysis of a hybrid DG method for Stokes flow, *IMA Journal of Numerical Analysis* 33 (2) (2013) 687–721.
- [24] H. Egger, C. Waluga, A hybrid discontinuous Galerkin method for Darcy-Stokes problems, in: R. Bank, M. Holst, O. Widlund, J. Xu (Eds.), *Domain Decomposition Methods in Science and Engineering XX*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 663–670. doi:10.1007/978-3-642-35275-1_79.
- [25] N. Nguyen, J. Peraire, B. Cockburn, A hybridizable discontinuous Galerkin method for the incompressible Navier-Stokes equations, *AIAA Paper* 2010-362.
- [26] N. C. Nguyen, J. Peraire, Hybridizable discontinuous Galerkin methods for partial differential equations in continuum mechanics, *Journal of Computational Physics* 231 (2012) 5955–5988.
- [27] N. C. Nguyen, J. Peraire, B. Cockburn, High-order implicit hybridizable discontinuous Galerkin methods for acoustics and elastodynamics, *Journal of Computational Physics* 230 (2011) 3695–3718.
- [28] J. Schütz, G. May, A hybrid mixed method for the compressible Navier-Stokes equations, *Journal of Computational Physics* 240 (2013) 58–75.
- [29] N. C. Nguyen, J. Peraire, B. Cockburn, Hybridizable discontinuous Galerkin methods for the time-harmonic Maxwell’s equations, *Journal of*

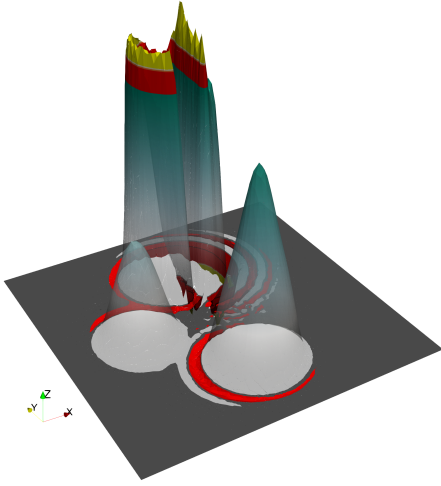
- Computational Physics 230 (19) (2011) 7151–7175. doi:10.1016/j.jcp.2011.05.018.
- [30] N. C. Nguyen, J. Peraire, B. Cockburn, An implicit high-order hybridizable discontinuous Galerkin method for nonlinear convection-diffusion equations, *Journal of Computational Physics* 228 (2009) 8841–8855.
 - [31] N. C. Nguyen, J. Peraire, B. Cockburn, An implicit high-order hybridizable discontinuous Galerkin method for linear convection-diffusion equations, *Journal of Computational Physics* 228 (2009) 3232–3254.
 - [32] H. Egger, J. Schöberl, A hybrid mixed discontinuous Galerkin finite element method for convection-diffusion problems, *IMA Journal of Numerical Analysis* 30 (2010) 1206–1234.
 - [33] T. Bui-Thanh, From Godunov to a unified hybridized discontinuous Galerkin framework for partial differential equations, *Journal of Computational Physics* 295 (2015) 114–146. doi:10.1016/j.jcp.2015.04.009.
 - [34] F. Frank, B. Reuter, V. Aizinger, P. Knabner, FESTUNG: A MATLAB/GNUOctave toolbox for the discontinuous Galerkin method. Part I: Diffusion operator, *Computers and Mathematics with Applications* 70 (1) (2015) 11 – 46. doi:10.1016/j.camwa.2015.04.013.
 - [35] B. Reuter, V. Aizinger, M. Wieland, F. Frank, P. Knabner, FESTUNG: A MATLAB/GNU Octave toolbox for the discontinuous Galerkin method. Part II: Advection operator and slope limiting, *Computers and Mathematics with Applications* 72 (7) (2016) 1896–1925. doi:10.1016/j.camwa.2016.08.006.
 - [36] B. Reuter, F. Frank, V. Aizinger, FESTUNG—The Finite Element Simulation Toolbox for UNstructured Grids (2017). doi:10.5281/zenodo.839054.
URL <https://www.math.fau.de/FESTUNG>
 - [37] B. Reuter, F. Frank, FESTUNG: The Finite Element Simulation Toolbox for UNstructured Grids (2017).
URL <https://github.com/FESTUNG>
 - [38] D. Kuzmin, A vertex-based hierarchical slope limiter for adaptive discontinuous Galerkin methods, *Journal of Computational and Applied Mathematics* 233 (12) (2010) 3077–3085, *Finite Element Methods in Engineering and Science (FEMTEC 2009)*. doi:10.1016/j.cam.2009.05.028.
 - [39] V. Aizinger, A geometry independent slope limiter for the discontinuous Galerkin method, in: E. Krause, Y. Shokin, M. Resch, D. Kröner, N. Shokina (Eds.), *Computational Science and High Performance Computing IV*, Vol. 115 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, Springer Berlin Heidelberg, 2011, pp. 207–217. doi:10.1007/978-3-642-17770-5_16.
 - [40] B. Reuter, A. Rupp, V. Aizinger, P. Knabner, FESTUNG: A MATLAB/GNUOctave toolbox for the discontinuous Galerkin method. Part IV: Generic problem framework and model coupling interface, in preparation.
 - [41] E. Godlewski, P.-A. Raviart, *Hyperbolic Systems of Conservation Laws*, Ellipses Paris, 1991.
 - [42] E. Godlewski, P.-A. Raviart, *Numerical Approximation of Hyperbolic Systems of Conservation Laws*, Springer New York, 1996.
 - [43] D. Kröner, *Numerical Schemes for Conservation Laws*, Wiley Teubner, 1997.
 - [44] P. Lax, Weak solutions of nonlinear hyperbolic equations and their numerical computation, *Communications on Pure and Applied Mathematics* 7 (1954) 159–193.
 - [45] V. V. Rusanov, The calculation of the interaction of non-stationary shock waves with barriers, *Journal of Computational Mathematics and Mathematical Physics (in Russian)* 1 (1961) 267–279.
 - [46] R. Alexander, Diagonally implicit Runge-Kutta methods for stiff O.D.E.'s., *SIAM Journal of Numerical Analysis* 14 (1977) 1006–1021.
 - [47] E. Hairer, G. Wanner, *Solving Ordinary Differential Equations II*, Springer Series in Computational Mathematics, 1991.
 - [48] A. Jaust, J. Schütz, A temporally adaptive hybridized discontinuous Galerkin method for time-dependent compressible flows, *Computers and Fluids* 98 (2014) 177–185. doi:10.1016/j.compfluid.2014.01.019.
 - [49] R. J. LeVeque, High-resolution conservative algorithms for advection in incompressible flow, *SIAM Journal on Numerical Analysis* 33 (2) (1996) 627–665. doi:10.2307/2158391.
 - [50] S. Badia, J. Bonilla, A. Hierro, Differentiable monotonicity-preserving schemes for discontinuous Galerkin methods on arbitrary meshes, *Computer Methods in Applied Mechanics and Engineering* 320 (2017) 582 – 605. doi:10.1016/j.cma.2017.03.032.



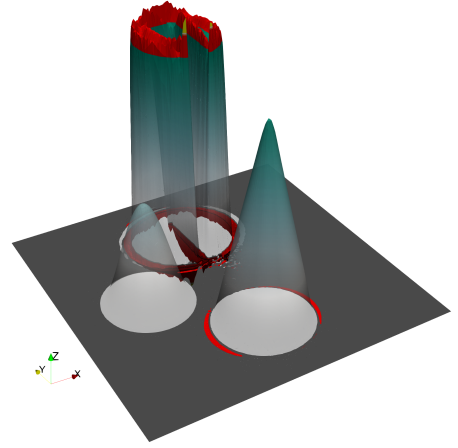
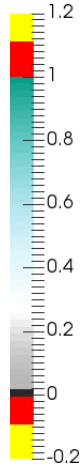
(a) Exact solution.



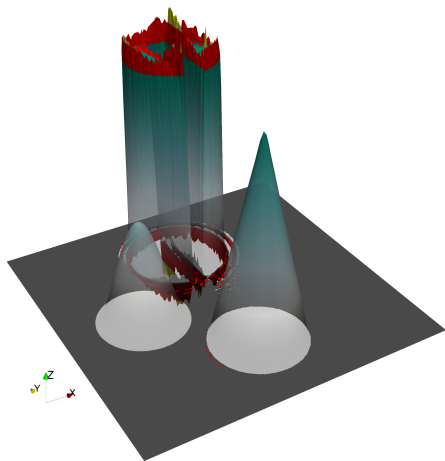
(b) $p = 0$.



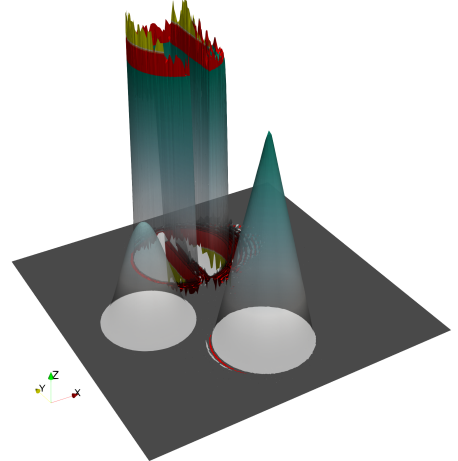
(c) $p = 1$.



(d) $p = 2$.

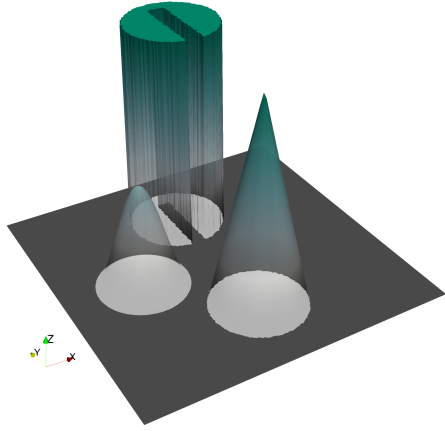


(e) $p = 3$.

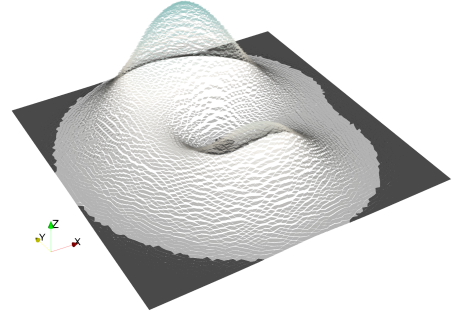


(f) $p = 4$.

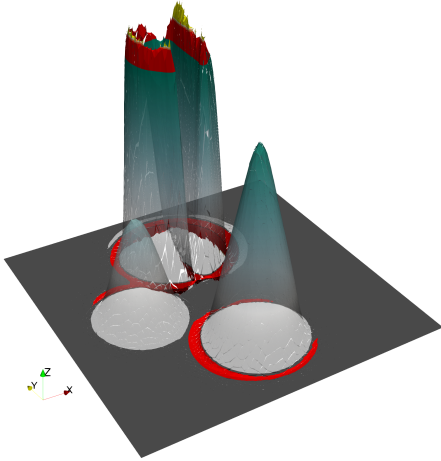
Figure 3: HDG solutions for the solid body rotation benchmark for different polynomial orders at end time $t_{\text{end}} = 2\pi$.



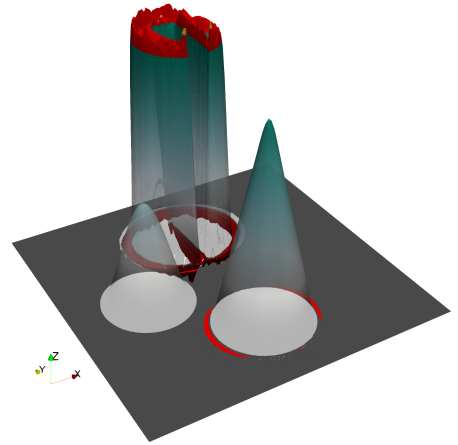
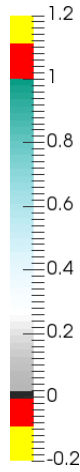
(a) Exact solution.



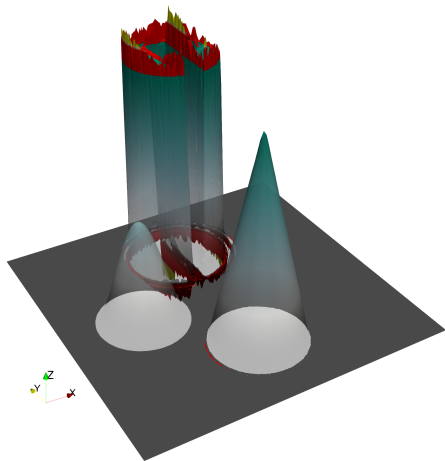
(b) $p = 0$.



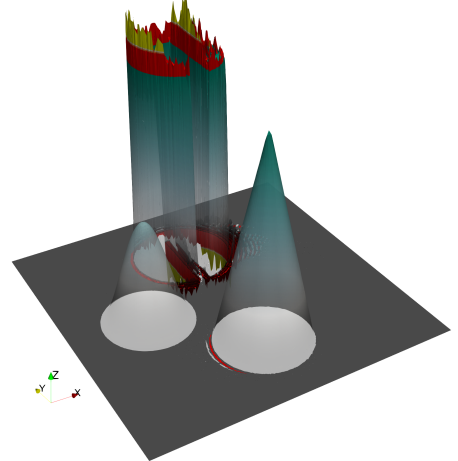
(c) $p = 1$.



(d) $p = 2$.



(e) $p = 3$.



(f) $p = 4$.

Figure 4: DG solutions for the solid body rotation benchmark for different polynomial orders at end time $t_{\text{end}} = 2\pi$.

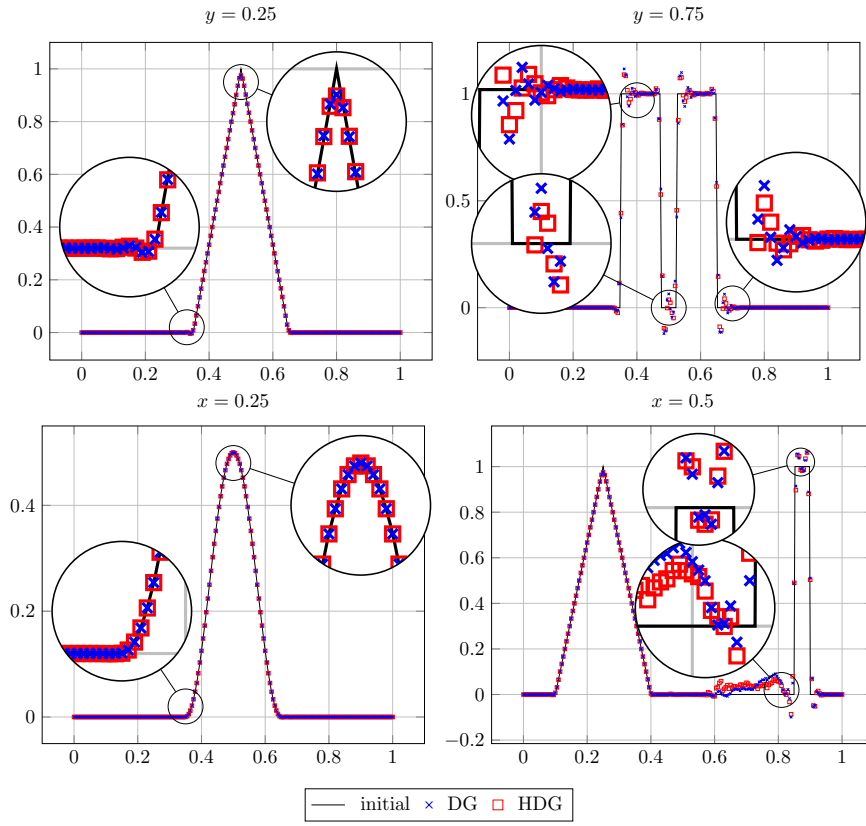


Figure 5: Cross-sections of the DG solutions for the solid body rotation benchmark with $p = 3$ at end time $t_{\text{end}} = 2\pi$.