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

*Alexander Jaust, Balthasar Reuter, Vadym Aizinger, Jochen Schütz and Peter Knabner*

UHasselt Computational Mathematics Preprint
Nr. UP-17-09

August 19, 2017

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

Alexander Jaust*[1], Balthasar Reuter†[2], Vadym Aizinger‡[3,2], Jochen Schütz§[1] and Peter Knabner¶[2]

[1]UHasselt – Hasselt University, Faculty of Sciences, Agoralaan Gebouw D, 3590 Diepenbeek, Belgium
[2]Friedrich–Alexander University of Erlangen–Nürnberg, Department of Mathematics, Cauerstraße 11, 91058 Erlangen, Germany
[3]Alfred Wegener Institute, Helmholtz Centre for Polar and Marine Research, Am Handelshafen 12, 27570 Bremerhaven, Germany

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 was 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 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.

## 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 fits extremely well [6] into the popular parallel and hybrid computational paradigms.

*alexander.jaust@uhasselt.be
†reuter@math.fau.de
‡vadym.aizinger@awi.de, corresponding author
§jochen.schuetz@uhasselt.be
¶knabner@math.fau.de

However, one aspect of the DG methodology places it at clear disadvantage compared to classical finite element and finite volume approaches: 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 classical idea to overcome this issue—and in many cases even to speed up the time-to-solution [7, 8, 9]—is to use *hybridization*. Roughly speaking, this means that the discretized PDE is augmented with an unknown, let us call it $\lambda_h$ 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 a 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 [10], it has subsequently been used in the context of mixed methods [11, 12]. In those works, $\lambda_h$ 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 [13], Cockburn and coworkers introduced the hybridized discontinuous Galerkin method in a unifying framework in [14]. Subsequently, the method has been extended to various types of equations such as the Stokes [15, 16] and Darcy-Stokes equations [17], the incompressible and compressible Navier-Stokes equations [18, 19, 20, 21], the Maxwell equations [22], and—particularly relevant for this work—to the convection(-diffusion) equation [23, 24, 25]. For an interesting unification framework, we would also like to draw reader's attention to a recent publication [26].

The current work applies and extends to the hybridized schemes the framework and design principles introduced in [27, 28] for "standard" DG formulations and epitomized in our MATLAB / GNU Octave toolbox *FESTUNG* (*F*inite *E*lement *S*imulation *T*oolbox for *UN*structured *G*rids) available at [29, 30]. Citing from [27], our chief goals are:

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 [27], 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 [31, 32] is presented in the second paper in series [28]. 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 [33].

The rest of the paper is structured as follows: Our model problem is introduced in the next section accompanied by a detailed description of the space and time discretization in Section 3. We present important aspects of the implementation in Section 4. In Section 5, we verify the code by means of analytical convergence tests and compare numerical results to those of the unhybridized discontinuous Galerkin implementation of the model problem from our previous publication [28]. 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 $J \coloneqq (0, t_{\text{end}})$ be a finite time interval and $\Omega \subset \mathbb{R}^2$ a polygonally bounded, Lipschitz domain with boundary $\partial\Omega$. We solve the *linear advection equation*

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

where the scalar quantity $c : J \times \Omega \to \mathbb{R}$ is unknown, and the flux function $\mathbf{f} : J \times \Omega \times \mathbb{R} \to \mathbb{R}^2$ 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

$$\mathbf{f}\,(t, \mathbf{x}, c\,(t, \mathbf{x})) \coloneqq \begin{bmatrix} u^1(t, \mathbf{x}) & u^2(t, \mathbf{x}) \end{bmatrix}^{\mathrm{T}} c\,(t, \mathbf{x}) \tag{2}$$

with given velocity $\mathbf{u} \coloneqq [u^1, u^2]^{\mathrm{T}}$ and source function $h : J \times \Omega \to \mathbb{R}$ independent of $c$. Additionally, the equation is equipped with some initial condition

$$c \;=\; c^0 \quad \text{on } \{0\} \times \Omega$$

and Dirichlet boundary condition

$$c \;=\; c_{\mathrm{D}} \quad \text{on } J \times \partial\Omega_{\text{in}}(t)$$

on inflow boundaries $\partial\Omega_{\text{in}}(t) \coloneqq \{\mathbf{x} \in \partial\Omega \,|\, \mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu}(\mathbf{x}) < 0\}$ with $\boldsymbol{\nu}(\mathbf{x})$ denoting the outward pointing normal vector. No boundary conditions need to be prescribed on the outflow boundary $\partial\Omega_{\text{out}}(t) \coloneqq \partial\Omega \setminus \Omega_{\text{in}}(t)$. Functions $c^0 : \Omega \mapsto \mathbb{R}$ and $c_D : J \times \partial\Omega_{\text{in}}(t) \mapsto \mathbb{R}$ are known. The problem is the same as the one considered in the second paper of our series [28] with a slightly different notation.

# 3 Discretization

## 3.1 Notation

Let $\mathcal{T}_h = \{T\}$ be a regular family of non-overlapping partitions of a polygonally bounded domain $\Omega$ 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 $\mathcal{E}_h = \mathcal{E}_{\text{int}} \cup \mathcal{E}_{\text{bc}}$ consisting of sets of interior edges $\mathcal{E}_{\text{int}}$ and boundary edges $\mathcal{E}_{\text{bc}}$. The latter is split into inflow $\mathcal{E}_{\text{in}}$ and outflow edges $\mathcal{E}_{\text{out}}$. 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 $T_{\bar{k}^-}$. Boundary edges have only one adjacent element and, for interior edges, the element opposite to $T_{\bar{k}^-}$ is called $T_{\bar{k}^+}$. The total number of edges is denoted by $\overline{K} \coloneqq |\mathcal{E}_h|$, and $\Gamma = \bigcup_{\bar{k}=1}^{\overline{K}} E_{\bar{k}}$ is the trace of the mesh. We refer to elements using index $k$ and add bars $\bar{\cdot}$ (e.g., $\bar{k}$ or $\overline{K}$) whenever we refer to edges to distinguish one from another. Additionally, we will make use of the notation from previous publications [27, 28] 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").

For the description of the method, we need mappings that allow 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 $\mathcal{E}_h$

$$\rho : \{0, 1, \ldots, K\} \times \{1, 2, 3\} \ni (k, n) \mapsto \bar{k} \in \{1, \ldots \overline{K}\}\,. \tag{3}$$

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., $E_{\bar{k}} = E_{k^-n^-} = E_{k^+n^+}$. 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 $T_{\bar{k}^+}$ that exists only for interior edges $\mathcal{E}_{\text{int}}$,

$$\kappa : \{1, \dots, \overline{K}\} \times \{1, 2\} \ni (k, l) \mapsto \begin{Bmatrix} k^- \in \{1, \dots, K\}, & \text{if } l = 1 \\ k^+ \in \{0, \dots, K\}, & \text{if } l = 2 \end{Bmatrix} . \tag{4}$$

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

## 3.2 Semi-discrete form

Given $\mathbb{P}_p(T)$ and $\mathbb{P}_p(E)$, the spaces of complete polynomials of degree at most $p$ on an element $T \in \mathcal{T}_h$ or edge $E \in \mathcal{E}_h$, we denote broken polynomial spaces $V_h, M_h$ on the triangulation $\mathcal{T}_h$ and its set of edges $\mathcal{E}_h$, respectively,

$$V_h \coloneqq \{\varphi_h : \bar{\Omega} \to \mathbb{R} : \ \forall T \in \mathcal{T}_h, \ \varphi_h|_T \in \mathbb{P}_p(T)\}, \qquad M_h \coloneqq \{\mu_h : \Gamma \to \mathbb{R} : \ \forall E \in \mathcal{E}_h, \ \mu_h|_E \in \mathbb{P}_p(E)\}. \tag{5}$$

We obtain a weak formulation by multiplying (1) with a test function $\varphi_h \in V_h$, applying integration by parts on element $T_k \in \mathcal{T}_h$, and replacing the flux $\mathbf{f}$ on element boundaries $\partial T_k$ by a numerical flux function

$$\hat{\mathbf{f}}(\lambda_h, c_h^-) \coloneqq \mathbf{f}(\lambda_h) - \alpha(\lambda_h - c_h^-)\, \boldsymbol{\nu} \tag{6}$$

introducing in the process a new unknown $\lambda_h$ defined on edges; this along with an additional equation that ensures the flux over element boundaries in normal direction is conservative yields

$$\int_{T_k} \partial_t c_h \, \varphi_h \, \mathrm{d}\mathbf{x} - \int_{T_k} \mathbf{f}(c_h) \cdot \nabla \varphi_h \, \mathrm{d}\mathbf{x} + \int_{\partial T_k} \hat{\mathbf{f}}(\lambda_h, c_h^-) \cdot \boldsymbol{\nu} \, \varphi_h^- \, \mathrm{d}s = \int_{T_k} h \, \varphi_h \, \mathrm{d}\mathbf{x}, \qquad \forall \varphi_h \in V_h, \tag{7a}$$

$$\int_{E_{\bar{k}} \in \mathcal{E}_{\text{int}}} \alpha \mu_h \left(2\lambda_h - c_h^- - c_h^+\right) \mathrm{d}s + \int_{E_{\bar{k}} \in \mathcal{E}_{\text{bc}}} \mu_h \left(\lambda_h - c_{\partial\Omega}(c_h^-)\right) \mathrm{d}s = 0, \qquad \forall \mu_h \in M_h. \tag{7b}$$

Due to the local nature of the DG method, we can formulate the variational equations on a triangle-by-triangle and edge-by-edge basis, respectively. The numerical flux (6) is a modified Lax-Friedrichs/Rusanov flux with a stabilization parameter $\alpha > 0$ that must be chosen t least of the same magnitude as the largest (in absolute value) eigenvalue of the system representing the local normal flux. Substituting the numerical flux into (7) leads to the semi-discrete system

$$\begin{aligned} \int_{T_k} \partial_t c_h \, \varphi_h \, \mathrm{d}\mathbf{x} - \int_{T_k} \mathbf{f}(c_h) \cdot \nabla \varphi_h \, \mathrm{d}\mathbf{x} + \int_{\partial T_k \setminus \partial\Omega} \left(\mathbf{f}(\lambda_h) \cdot \boldsymbol{\nu} - \alpha(\lambda_h - c_h^-)\right) \varphi_h^- \, \mathrm{d}s \\ + \int_{\partial T_k \cap \partial\Omega} \left(\mathbf{f}\big(c_{\partial\Omega}(\lambda_h)\big) \cdot \boldsymbol{\nu}\right) \varphi_h^- \, \mathrm{d}s = \int_{T_k} h \, \varphi_h \, \mathrm{d}\mathbf{x}, \quad \forall \varphi_h \in V_h, \end{aligned} \tag{8a}$$

$$\int_{E_{\bar{k}} \in \mathcal{E}_{\text{int}}} \alpha \mu_h \left(2\lambda_h - c_h^- - c_h^+\right) \mathrm{d}s + \int_{E_{\bar{k}} \in \mathcal{E}_{\text{bc}}} \mu_h \left(\lambda_h - c_{\partial\Omega}(c_h^-)\right) \mathrm{d}s = 0, \qquad \forall \mu_h \in M_h. \tag{8b}$$

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 $\lambda_h$ are apparent. However, the choice of the numerical flux on element boundaries leads to an inter-element coupling solely by the function $\lambda_h$ and thus no longer depending on solution $c_h^+$ of the neighboring element. In Sec. 3.4, we explain 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.

### 3.2.1 Local basis representation

The DG function spaces defined in (5) do not have any continuity constraints across element or edge boundaries. Consequently, a two-dimensional basis function $\varphi_{ki} : \bar{\Omega} \to \mathbb{R}$ for $V_h$ is only supported on $T_k \in \mathcal{T}_h$ and must fulfill

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

In the same way, a one-dimensional basis function $\mu_{\bar{k}i} : \Gamma \to \mathbb{R}$ for $M_h$ is only supported on the edge $E_{\bar{k}} \subset \Gamma$, ensuring

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

We denote the numbers of local degrees of freedom on a triangle or an edge by $N$ or $\overline{N}$, 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 $\varphi_{ki}$ we refer to our first paper [27]. The one-dimensional basis functions up to order four given on the unit interval $[0, 1]$ are defined as

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

The approximated local solutions $c_h$ on $T_k \in \mathcal{T}_h$ and $\lambda_h$ on $E_{\bar{k}} \in \mathcal{E}_h$ are represented using the local basis on elements and edges

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

Note that we do not encode a time dependency in $\lambda_h$ as it only implicitly depends on time through $c_h$.

### 3.2.2 System of equations

Testing (8a) with $\varphi_h = \varphi_{ki}$ yields a time-dependent system of equations with the contribution from $T_k$ given by

$$\begin{aligned}
&\underbrace{\partial_t \sum_{j=1}^{N} C_{kj}(t) \int_{T_k} \varphi_{kj}\, \varphi_{ki}\, \mathrm{d}\mathbf{x}}_{I\ (\mathbf{M}_\varphi)} - \underbrace{\sum_{j=1}^{N} C_{kj}(t) \sum_{m=1}^{2} \int_{T_k} u^m(t, \mathbf{x})\, \varphi_{kj}\, \partial_{x^m} \varphi_{ki}\, \mathrm{d}\mathbf{x}}_{II\ (-\mathbf{G}^1 - \mathbf{G}^2)}\\
&\quad + \underbrace{\sum_{j=1}^{\overline{N}} \Lambda_{kj} \int_{\partial T_k \setminus \partial\Omega} (\mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu})\ \mu_{\bar{k}j}\, \varphi_{ki}\, \mathrm{d}s}_{III\ (\mathbf{S})} - \underbrace{\alpha \sum_{j=1}^{\overline{N}} \Lambda_{kj} \int_{\partial T_k \setminus \partial\Omega} \mu_{\bar{k}j}\, \varphi_{ki}\, \mathrm{d}s}_{IV\ (\alpha\mathbf{R}_\mu)} + \underbrace{\alpha \sum_{j=1}^{N} C_{kj} \int_{\partial T_k \setminus \partial\Omega} \varphi_{kj}\, \varphi_{ki}\, \mathrm{d}s}_{V\ (\alpha\mathbf{R}_\varphi)}\\
&\quad + \underbrace{\int_{\partial T_k \cap \partial\Omega} \varphi_{ki}\, (\mathbf{u}(t, \mathbf{x}) \cdot \boldsymbol{\nu}) \begin{Bmatrix} \sum_{j=1}^{\overline{N}} \Lambda_{kj}\, \mu_{\bar{k}j} & \text{on } \partial\Omega_{\text{out}} \\ c_{\text{D}}(t, \mathbf{x}) & \text{on } \partial\Omega_{\text{in}} \end{Bmatrix} \mathrm{d}s}_{VI\ (\mathbf{S}_{\text{out}}, \mathbf{F}_{\varphi,\text{in}})} = \underbrace{\int_{T_k} h\, \varphi_{ki}\, \mathrm{d}\mathbf{x}}_{VII\ (\mathbf{H})}\,.
\end{aligned} \tag{10a}$$

We already use the assumption that $\mathbf{f}$ is a linear function. Additionally, the semi-discrete form of (8b) is given by

$$\begin{aligned}
&\underbrace{\alpha \sum_{j=1}^{\overline{N}} \Lambda_{kj} \int_{\partial T_k \setminus \partial\Omega} \mu_{\bar{k}j}\, \mu_{\bar{k}i}\, \mathrm{d}s}_{VIII\ (\alpha\bar{\mathbf{M}}_\mu)} - \underbrace{\alpha \sum_{j=1}^{N} C_{kj} \int_{\partial T_k \setminus \partial\Omega} \varphi_{kj}\, \mu_{\bar{k}i}\, \mathrm{d}s}_{IX\ (-\alpha\mathbf{T})}\\
&\quad + \underbrace{\sum_{j=1}^{\overline{N}} \Lambda_{kj} \int_{\partial T_k \cap \partial\Omega} \mu_{\bar{k}j}\, \mu_{\bar{k}i}\, \mathrm{d}s}_{X\ (\tilde{\mathbf{M}}_\mu)} - \underbrace{\int_{\partial T_k \cap \partial\Omega} \mu_{\bar{k}i} \begin{Bmatrix} \sum_{j=1}^{N} C_{kj}\, \varphi_{kj} & \text{on } \partial\Omega_{\text{out}} \\ c_D(t, \mathbf{x}) & \text{on } \partial\Omega_{\text{in}} \end{Bmatrix} \mathrm{d}s}_{XI\ (-\mathbf{K}_{\mu,\text{out}} \text{ and } \mathbf{K}_{\mu,\text{in}})} = 0
\end{aligned} \tag{10b}$$

We can rewrite system (10) in matrix form

$$\mathbf{M}_\varphi \partial_t \mathbf{C} + \underbrace{\left(-\mathbf{G}^1 - \mathbf{G}^2 + \alpha \mathbf{R}_\varphi\right)}_{=:\tilde{\mathbf{L}}(t)} \mathbf{C} + \underbrace{\left(\mathbf{S} + \mathbf{S}_{\text{out}} - \alpha \mathbf{R}_\mu\right)}_{=:\overline{\mathbf{M}}(t)} \mathbf{\Lambda} = \underbrace{\mathbf{H} - \mathbf{F}_{\varphi,\text{in}}}_{=:\mathbf{B}_\varphi(t)}, \tag{11a}$$

$$\underbrace{\left(-\alpha \mathbf{T} - \mathbf{K}_{\mu,\text{out}}\right)}_{=:\mathbf{N}} \mathbf{C} + \underbrace{\left(\alpha \overline{\mathbf{M}}_\mu + \tilde{\mathbf{M}}_\mu\right)}_{=:\mathbf{P}} \mathbf{\Lambda} = \underbrace{-\mathbf{K}_{\mu,\text{in}}}_{=:\mathbf{B}_\mu(t)} \tag{11b}$$

with representation vectors

$$\mathbf{C}(t) \coloneqq \left[C_{11}(t) \cdots C_{1N}(t) \cdots \cdots C_{K1}(t) \cdots C_{KN}(t)\right]^{\mathrm{T}}, \qquad \mathbf{\Lambda} \coloneqq \left[\Lambda_{11} \cdots \Lambda_{1\overline{N}} \cdots \cdots \Lambda_{\overline{K}1} \cdots \Lambda_{\overline{K}\overline{N}}\right]^{\mathrm{T}}.$$

At this point, we emphasize once more that inter-element coupling of the solution $c_h$ in system (11) acts only through the edge function $\lambda_h$ by means of $\overline{\mathbf{M}}(t) \in \mathbb{R}^{KN \times \overline{K}\overline{N}}$ and $\mathbf{N} \in \mathbb{R}^{\overline{K}\overline{N} \times KN}$. As a result, these block matrices have rectangular blocks and are usually rectangular themselves. This stems from the fact that the numbers of local degrees of freedom on elements $N$ and edges $\overline{N}$ differ for $p > 0$ (i. e., $N > \overline{N}$) leading to rectangular blocks in the matrices, and the number of edges $\overline{K}$ differs from the number of elements $K$.

### 3.2.3 Contributions from volume terms I, II and VI

All matrices in system (11) 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

$$[\mathbf{M}_\varphi]_{(k-1)N+i,(k-1)N+j} \coloneqq \int_{T_k} \varphi_{kj}\, \varphi_{ki} \,\mathrm{d}\mathbf{x}\,. \tag{12a}$$

This leads to a block diagonal matrix because basis and test functions $\varphi_{ki}$, $i \in \{1, \ldots, N\}$ have support only on each element $T_k$. Therefore

$$\mathbf{M}_\varphi = \begin{bmatrix} \mathbf{M}_{\varphi,T_1} & & \\ & \ddots & \\ & & \mathbf{M}_{\varphi,T_N} \end{bmatrix} \quad \text{with } \textit{local mass matrix} \quad \mathbf{M}_{\varphi,T_k} \coloneqq \int_{T_k} \begin{bmatrix} \varphi_{k1}\varphi_{k1} & \cdots & \varphi_{k1}\varphi_{kN} \\ \vdots & \ddots & \vdots \\ \varphi_{kN}\varphi_{k1} & \cdots & \varphi_{kN}\varphi_{kN} \end{bmatrix} \mathrm{d}\mathbf{x} \in \mathbb{R}^{N \times N}. \tag{12b}$$

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

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 [28] 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

$$[\mathbf{G}^m]_{(k-1)N+i,(k-1)N+j} = \int_{T_k} u^m\, \varphi_{kj}\, \partial_{x^m} \varphi_{ki} \,\mathrm{d}\mathbf{x} \tag{13a}$$

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

$$\mathbf{G}^m_{T_K} \coloneqq \int_{T_k} u^m \begin{bmatrix} \varphi_{k1}\partial_{x^m}\varphi_{k1} & \cdots & \varphi_{kN}\,\partial_{x^m}\varphi_{k1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1}\,\partial_{x^m}\varphi_{kN} & \cdots & \varphi_{kN}\,\partial_{x^m}\varphi_{kN} \end{bmatrix} \mathrm{d}\mathbf{x}\,. \tag{13b}$$

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

$$[\mathbf{H}]_{(k-1)N+i} = \int_{T_k} h\, \varphi_{ki} \,\mathrm{d}\mathbf{x} \tag{14a}$$

such that the full vector is easily assembled as

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{T_1} \\ \vdots \\ \mathbf{H}_{T_K} \end{bmatrix} \qquad \text{with} \quad \mathbf{H}_{T_k} = \int_{T_k} h \begin{bmatrix} \varphi_{k1} \\ \vdots \\ \varphi_{kN} \end{bmatrix} \mathrm{d}\mathbf{x}. \tag{14b}$$

#### 3.2.4 Contribution of edge terms Ⅱ, Ⅳ, Ⅴ, Ⅵ— first equation

Compared to the DG discretization used in our previous works [27, 28], the number of edge integrals has significantly increased. This is related to the following factors:

1. Edge integrals are split into integrals over interior edges $\int_{\partial T_k \setminus \partial\Omega}$ and over edges on the domain boundary $\int_{\partial T_k \cap \partial\Omega}$.
2. Numerical flux function (6) introduces three terms compared to only one for the upwind flux used in [28].
3. An additional edge unknown is introduced that only exists on edges resulting in additional equation (8b).

To improve readability we split the presentation of edge integrals into two sections: edge terms in the original equation for $c_h$ are explained in the following, followed by the edge terms in the hybrid equation. Throughout the assembly description and within the implementation we use the *element-based view*, i.e., we present all edge terms in a form, which would allow to assemble them 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 them in a single loop over all edges $E_{\bar{k}}$, $\bar{k} \in \{1, \dots, \overline{K}\}$. We made this choice since the data structures in FESTUNG favor the element-based view. For that reason, from now on, we always consider $\mu_{knj} = \mu_{\bar{k}j}$ with the mapping from $(k, n)$ to $\bar{k}$ given in (3).

Term Ⅲ contributes matrix $\mathsf{S} \in \mathbb{R}^{KN \times \overline{K}\overline{N}}$ given as

$$[\mathsf{S}]_{(k-1)N+i,(\bar{k}-1)\overline{N}+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\mathrm{int}}} \int_{E_{kn}} (\mathbf{u} \cdot \boldsymbol{\nu}_{kn})\, \mu_{knj}\, \varphi_{ki} \, \mathrm{d}s \,. \tag{15a}$$

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

$$\mathsf{S}_{E_{kn}} = \int_{E_{kn}} (\mathbf{u} \cdot \boldsymbol{\nu}_{kn}) \begin{bmatrix} \mu_{kn1}\, \varphi_{k1} & \cdots & \mu_{kn\overline{N}}\, \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \mu_{kn1}\, \varphi_{kN} & \cdots & \mu_{kn\overline{N}}\, \varphi_{kN} \end{bmatrix} \mathrm{d}s \,. \tag{15b}$$

Note that this is the contribution of a single interior edge $E_{kn} \in \partial T_k \cap \mathcal{E}_{\mathrm{int}}$ of triangle $T_k$. As this integral is also evaluated on the neighboring element, this block will end up in the matrix *twice*.

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

$$[\mathsf{R}_\mu]_{(k-1)N+i,(\bar{k}-1)\overline{N}+j} = \sum_{E_{kn} \in \partial T_k \cap \mathcal{E}_{\mathrm{int}}} \int_{E_{kn}} \mu_{knj}\, \varphi_{ki} \, \mathrm{d}s \tag{16a}$$

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

$$\mathsf{R}_{\mu,E_{kn}} = \int_{E_{kn}} \begin{bmatrix} \mu_{kn1}\, \varphi_{k1} & \cdots & \mu_{kn\overline{N}}\, \varphi_{k1} \\ \vdots & \ddots & \vdots \\ \mu_{kn1}\, \varphi_{kN} & \cdots & \mu_{kn\overline{N}}\, \varphi_{kN} \end{bmatrix} \mathrm{d}s \,. \tag{16b}$$

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

$$[\mathbf{R}_\varphi]_{(k-1)N+i,(k-1)N+j} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{int}}} \int_{E_{kn}} \varphi_{kj}\,\varphi_{ki}\,\mathrm{d}s. \tag{17a}$$

The element-local matrix is then

$$\mathbf{R}_{\varphi,T_k} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{int}}} \int_{E_{kn}} \begin{bmatrix} \varphi_{k1}\,\varphi_{k1} & \cdots & \varphi_{kN}\,\varphi_{k1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1}\,\varphi_{kN} & \cdots & \varphi_{kN}\,\varphi_{kN} \end{bmatrix} \mathrm{d}s\,, \tag{17b}$$

where each edge $E_{kn}$ of each triangle $T_k$ is visited exactly once, and $\mathbf{R}_\varphi = \operatorname{diag}(\mathbf{R}_{\varphi,T_1},\dots,\mathbf{R}_{\varphi,T_K})$. This is slightly different from previous works [28] where the test functions $\varphi^-$ and $\varphi^+$ 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 $E_{\bar{k}} \in \mathcal{E}_{\text{bc}}$. 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

$$[\mathbf{F}_{\varphi,\text{in}}]_{(k-1)N+i} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{in}}} \int_{E_{kn}} (\mathbf{u}\cdot\boldsymbol{\nu}_{kn})\,c_{\text{D}}\,\varphi_{ki}\,\mathrm{d}s\,. \tag{18}$$

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

$$[\mathbf{S}_{\text{out}}]_{(k-1)N+i,(\bar{k}-1)\overline{N}+j} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{out}}} \int_{E_{kn}} (\mathbf{u}\cdot\boldsymbol{\nu}_{kn})\,\mu_{knj}\,\varphi_{ki}\,\mathrm{d}s\,. \tag{19}$$

This is almost identical to (15a) with the only difference being the set of edges considered, and thus the sub-blocks take the same form given in equation (15b). In the implementation, $\mathbf{S}$ and $\mathbf{S}_{\text{out}}$ 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 stability parameter $\alpha$ that has to be respected and the fact that every edge is visited twice because it is an integral over interior edges $E_{\bar{k}} \in \mathcal{E}_{\text{int}}$. $\overline{\mathbf{M}}_\mu \in \mathbb{R}^{\overline{K}\overline{N}\times\overline{K}\overline{N}}$ is given by

$$[\overline{\mathbf{M}}_\mu]_{(\bar{k}-1)\overline{N}+i,(\bar{k}-1)\overline{N}+j} \coloneqq \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{int}}} \int_{E_{kn}} \mu_{knj}\,\mu_{kni}\,\mathrm{d}s \tag{20}$$

for $i,j \in \{1,2,\dots,\overline{N}\}$ and $\bar{k} \in \{1,2,\dots,\overline{K}\}$. This leads to a block diagonal matrix because the ansatz and test functions $\mu_{kni} = \mu_{\bar{k}i}, i \in \{1,2,\dots,\overline{N}\}$ 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 $\mathbf{T} \in \mathbb{R}^{\overline{K}\overline{N}\times KN}$ is given as

$$[\mathbf{T}]_{(\bar{k}-1)\overline{N}+i,(k-1)N+j} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\text{int}}} \int_{E_{kn}} \varphi_{kj}\,\mu_{kni}\,\mathrm{d}s \tag{21a}$$

with $i \in \{1,\dots,\overline{N}\}$, $j \in \{1,\dots,N\}$, and $\bar{k}$ given through the mapping in Eq. (3). The contribution of a single edge is

$$\mathbf{T}_{E_{kn}} = \int_{E_{kn}} \begin{bmatrix} \varphi_{k1}\,\mu_{kn1} & \cdots & \varphi_{kN}\,\mu_{kn1} \\ \vdots & \ddots & \vdots \\ \varphi_{k1}\,\mu_{kn1} & \cdots & \varphi_{kN}\,\mu_{kn\overline{N}} \end{bmatrix} \mathrm{d}s. \tag{21b}$$

In fact, we have $\mathsf{T} = \mathsf{R}_\mu^{\mathrm{T}}$ from (16b). Term X gives us the edge mass matrix on boundary edges

$$[\tilde{\mathsf{M}}_\mu]_{(\bar{k}-1)\overline{N}+i,(\bar{k}-1)\overline{N}+j} \coloneqq \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\mathrm{bc}}} \int_{E_{kn}} \mu_{knj}\,\mu_{kni}\,\mathrm{d}s \tag{22}$$

for $i,j \in \{1,2,\ldots,\overline{N}\}$, $\bar{k} \in \{1,2,\ldots,\overline{K}\}$, and with the matrix entries given in Eq. (20). 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 get a contribution to the right hand side $\mathbf{K}_{\mu,\mathrm{in}} \in \mathbb{R}^{\overline{K}\overline{N}}$

$$[\mathbf{K}_{\mu,\mathrm{in}}]_{(\bar{k}-1)\overline{N}+i} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\mathrm{in}}} \int_{E_{kn}} c_{\mathrm{D}}\,\mu_{kni}\,\mathrm{d}s\,, \tag{23a}$$

and outflow boundaries add a contribution to the matrix $\mathsf{K}_{\mu,\mathrm{out}} \in \mathbb{R}^{\overline{K}\overline{N}\times KN}$ as

$$[\mathsf{K}_{\mu,\mathrm{out}}]_{(\bar{k}-1)\overline{N}+i,(k-1)N+j} = \sum_{E_{kn}\in\partial T_k\cap\mathcal{E}_{\mathrm{out}}} \int_{E_{kn}} \varphi_{kj}\,\mu_{kni}\,\mathrm{d}s \tag{23b}$$

meaning that we obtain a block matrix similar to $\mathsf{T}$ (cf. (21b)) from every outflow edge.

## 3.3 Time discretization

The system of equations in (11) can be reformulated matrix notation as

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

This is a first order differential algebraic equation [19], 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 [34, 35].

The time interval $[t^0, t^{\mathrm{end}}]$ is discretized into not necessarily equidistant points $t^n \in J$ with $t^0 = 0 < t^1 < t^2 < \ldots < t^{\mathrm{end}}$. The time step size of the $n$th time step is given by $\Delta t^n = t^{n+1} - t^n$, and we abbreviate vectors and matrices on $n$th time level as $\mathbf{C}^n \coloneqq \mathbf{C}(t^n)$, etc. Time step adaptation can be easily achieved in DIRK schemes with embedded error estimators, e.g., as carried out for HDG methods in [36].

Let the DIRK scheme have $s$ stages. Then for the update at $t^{n+1}$ one has to solve

$$\begin{aligned}
\mathsf{M}_\varphi \mathbf{C}^{(i)} &= \mathsf{M}_\varphi \mathbf{C}^n + \Delta t \sum_{j=1}^{i} a_{ij}\,\mathbf{R}_{\mathrm{RK}}\left(t^{(j)}, \mathbf{C}^{(j)}, \mathbf{\Lambda}^{(j)}\right),\\
\mathbf{0} &= \mathbf{B}_\mu^{(i)} - \mathsf{N}\mathbf{C}^{(i)} - \mathsf{P}\mathbf{\Lambda}^{(i)}, \qquad i = 1,\ldots,s
\end{aligned}$$

with $t^{(i)} = t^n + c_i \Delta t^n$ and each stage requiring 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. The solution at the next time level is then given as

$$\mathsf{M}_\varphi \mathbf{C}^{n+1} = \mathsf{M}_\varphi \mathbf{C}^n + \Delta t \sum_{j=1}^{s} b_j\,\mathbf{R}_{\mathrm{RK}}^{(j)}.$$

This requires inversion of the mass matrix $\mathsf{M}_\varphi$ which is cheap because it is a block diagonal matrix. Coefficients $a_{ij}, b_j, c_i$ are defined in the routine `rungeKuttaImplicit` and can be arranged as a Butcher tableau (see Table 1).

**Remark 1** *All of the employed DIRK schemes are A- and L-stable [35].*

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

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

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

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

*This also means, that the last update $C^{(s)}$ is actually the updated solution at the new time $t^{n+1}$. Therefore, we can set $C^{n+1} = C^{(s)}$ after the last stage without the need of reconstructing it from the other intermediate solutions $C^{(i)}, i = 1, \dots, s$.*

## 3.4 Static condensation

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

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

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

We can write this compactly as

$$\mathbf{L}\boldsymbol{C}^{(i)} + \mathbf{M}\mathbf{\Lambda}^{(i)} = \mathbf{Q}, \tag{24a}$$

$$\mathbf{N}\boldsymbol{C}^{(i)} + \mathbf{P}\mathbf{\Lambda}^{(i)} = \mathbf{B}_\mu^{(i)}. \tag{24b}$$

The first line (24a) refers to equation (8a), and the second line (24b) to the hybrid part (8b). 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. We substitute

$$\boldsymbol{C}^{(i)} = \mathbf{L}^{-1}(\mathbf{Q} - \mathbf{M}\mathbf{\Lambda}^{(i)}) \tag{25}$$

from (24a) into (24b) and obtain

$$\left(-\mathbf{N}\mathbf{L}^{-1}\mathbf{M} + \mathbf{P}\right)\mathbf{\Lambda}^{(i)} = \mathbf{B}_\mu^{(i)} - \mathbf{N}\mathbf{L}^{-1}\mathbf{Q}. \tag{26}$$

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 (26) using $\mathbf{L}^{-1}\mathbf{Q}$ and $\mathbf{L}^{-1}\mathbf{M}$ and solve for $\mathbf{\Lambda}^{(i)}$. Once $\mathbf{\Lambda}^{(i)}$ is known, we can update $\boldsymbol{C}^{(i)}$ by substituting the updated value of $\mathbf{\Lambda}^{(i)}$ into (25).

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 $128 \cdot 2^{-p}$ elements to be a good choice in our case (for hardware details see Section 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 [27]. For the sake of completeness, we briefly introduce transformation rules for element and edge integrals as described in some detail in previous works [27, 28] and emphasize differences due to edge unknown $\lambda_h$. Finally, we describe 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 $\hat{T} = \{[0,0]^{\mathrm{T}}, [1,0]^{\mathrm{T}}, [0,1]^{\mathrm{T}}\}$ to any $T_k = \{\mathbf{x}_{k1}, \mathbf{x}_{k2}, \mathbf{x}_{k3}\} \in \mathcal{T}_h$,

$$\mathbf{F}_k : \quad \hat{T} \ni \hat{\mathbf{x}} \mapsto \mathbf{B}_k \hat{\mathbf{x}} + \mathbf{x}_{k1} = \mathbf{x} \in T_k \,, \qquad \text{with} \quad \mathbb{R}^{2\times 2} \ni \mathbf{B}_k \coloneqq \left[\mathbf{x}_{k2} - \mathbf{x}_{k1} \,\middle|\, \mathbf{x}_{k3} - \mathbf{x}_{k1}\right] . \tag{27}$$

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

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

For functions $w : T_k \to \mathbb{R}$ and $\hat{w} : \hat{T} \to \mathbb{R}$, 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:

$$\nabla \;=\; \left(\hat{\nabla}\mathbf{F}_k\right)^{-\mathrm{T}} \hat{\nabla} \;=\; \frac{1}{2\,|T_k|} \begin{bmatrix} B_k^{22}\,\partial_{\hat{x}^1} - B_k^{21}\,\partial_{\hat{x}^2} \\ B_k^{11}\,\partial_{\hat{x}^2} - B_k^{12}\,\partial_{\hat{x}^1} \end{bmatrix} , \tag{28}$$

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

$$\int_{T_k} w(\mathbf{x})\,\mathrm{d}\mathbf{x} \;=\; \frac{|T_k|}{|\hat{T}|} \int_{\hat{T}} w \circ \mathbf{F}_k(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} \;=\; 2|T_k| \int_{\hat{T}} w \circ \mathbf{F}_k(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} \;=\; 2|T_k| \int_{\hat{T}} \hat{w}(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} \,, \tag{29a}$$

$$\int_{E_{kn}} w(\mathbf{x})\,\mathrm{d}\mathbf{x} \;=\; \frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} w \circ \mathbf{F}_k(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} \;=\; \frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} \hat{w}(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} \,. \tag{29b}$$

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

$$\hat{\boldsymbol{\gamma}}_1(s) = \begin{bmatrix} 1-s \\ s \end{bmatrix} , \quad \hat{\boldsymbol{\gamma}}_2(s) = \begin{bmatrix} 0 \\ 1-s \end{bmatrix} , \quad \hat{\boldsymbol{\gamma}}_3(s) = \begin{bmatrix} s \\ 0 \end{bmatrix} , \tag{30}$$

and use this to transform Eq. (29b) further

$$\frac{|E_{kn}|}{|\hat{E}_n|} \int_{\hat{E}_n} \hat{w}(\hat{\mathbf{x}})\,\mathrm{d}\hat{\mathbf{x}} = \frac{|E_{kn}|}{|\hat{E}_n|} \int_0^1 \hat{w} \circ \hat{\boldsymbol{\gamma}}_n(s)\,|\hat{\boldsymbol{\gamma}}_n'(s)|\,\mathrm{d}s = |E_{kn}| \int_0^1 \hat{w} \circ \hat{\boldsymbol{\gamma}}_n(s)\,\mathrm{d}s \,, \tag{31}$$

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

Edge integrals with basis functions from the an adjoining element and basis functions defined on the edge, e.g., $\int_{E_{kn}} \varphi_{ki}\, \mu_{\bar{k}j}\, \mathrm{d}\mathbf{x}$ with $\bar{k} = \rho(k,n)$ as defined in (3), are transformed according to transformation rules (29b), (31)

$$\int_{E_{kn}} \varphi_{ki}\, \mu_{knj}\, \mathrm{d}\mathbf{x} = \frac{|E_{kn}|}{\left|\hat{E}_n\right|} \int_{\hat{E}_k} \hat{\varphi}_i(\hat{\mathbf{x}})\, \mu_{knj} \circ \boldsymbol{F}_k(\hat{\mathbf{x}})\, \mathrm{d}\hat{\mathbf{x}} = |E_{kn}| \int_0^1 \hat{\varphi}_i \circ \hat{\boldsymbol{\gamma}}_n(s)\, \hat{\mu}_j \circ \hat{\beta}_{kn}(s)\, \mathrm{d}s\,, \tag{32}$$

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

$$\hat{\beta}_{kn}(s) = \begin{cases} s & \text{if } \kappa(\rho(k,n),1) = k\,, \\ 1-s & \text{if } \kappa(\rho(k,n),2) = k \end{cases} \tag{33}$$

with $\kappa(\rho(k,n),l) = \kappa(\bar{k},l)$ given in (4). This does not introduce any further terms into the equation since $|\hat{\beta}'_{kn}(s)| = 1$.

## 4.2 Numerical integration

Similarly to other publications in this series [27, 28], we approximate triangle and edge integrals using numerical quadrature rules. Since we transform all integrals on $T_k \in \mathcal{T}_h$ to the reference triangle $\hat{T}$ and all integrals on $E_{kn} \in \mathcal{E}$ to reference interval $[0,1]$ (cf. Sec. 4.1), it is sufficient to define the quadrature rules on $\hat{T}$ and $[0,1]$, respectively:

$$\int_{\hat{T}} \hat{f}(\hat{\mathbf{x}})\, \mathrm{d}\hat{\mathbf{x}} \;\approx\; \sum_{r=1}^{R} \omega_r\, \hat{f}(\hat{\mathbf{q}}_r)\,, \qquad \int_0^1 \hat{g}(s)\, \mathrm{d}s \;\approx\; \sum_{r=1}^{R} \omega_r\, \hat{g}(\hat{q}_r) \tag{34}$$

with $R$ *quadrature points* $\hat{\mathbf{q}}_r \in \hat{T}$, $\hat{q}_r \in [0,1]$ and *quadrature weights* $\omega_r \in \mathbb{R}$. Note that quadrature points, weights, and their number is different for one- and two-dimensional quadrature rules. However, from context it is always clear whether a one- or two-dimensional quadrature rule is applied, and thus we abstain from distinguishing between them by means of additional subscripts in favor of better readability. The *order* of a quadrature rule is the largest integer $s$ such that (34) is *exact* for polynomials $\hat{f} \in \mathbb{P}_s(\hat{T})$, $\hat{g} \in \mathbb{P}_s([0,1])$, respectively. For all our numerical experiments, we use quadrature rules of order $2p+1$ on both elements and edges. The rules used in the implementation are found in routines `quadRule2D` and `quadRule1D`. An overview of quadrature rules on triangles can be found in the "Encyclopaedia of Cubature Formulas" [37]. For edge integration, we rely on standard Gauss quadrature rules of required order.

## 4.3 Assembly

In this section, the vectorized assembly of the block matrices in (11) is outlined. For that, we transform the required terms to reference triangle $\hat{T}$ or reference interval $[0,1]$, respectively, and then evaluate them by numerical quadrature.

As in previous papers in series [27, 28], 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

$$\mathbf{A} \otimes \mathbf{B} \coloneqq [[\mathbf{A}]_{i,j}\, \mathbf{B}] \in \mathbb{R}^{m_a m_b \times n_a n_b}\,. \tag{35}$$

Additionally, we employ operation $\mathbf{A} \otimes_{\mathrm{V}} \mathbf{B}$ with $m_b = r\, m_a, r \in \mathbb{N}$ introduced in our last publication [28] as

$$\mathbf{A} \otimes_{\mathrm{V}} \mathbf{B} \coloneqq \left[[\mathbf{A}]_{i,j}\, [\mathbf{B}]_{(i-1)r:ir,:}\right] \in \mathbb{R}^{m_b \times n_a n_b}\,, \tag{36}$$

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

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

Some of the block-matrices in system (11) appeared in identical form in previous publications [27, 28]. For brevity, we abstain from reproducing the corresponding assembly steps here and refer to the existing descriptions. This is the case for matrix $\mathbf{R}_\varphi$ (identical to $\mathbf{S}^{\text{diag}}$ in [27]), source terms $\mathbf{H}$ (cf. any of the papers in series), and mass matrix $\mathbf{M}_\varphi$. However, we use the latter to illustrate the basic idea of vectorized assembly and thus describe it here once again.

### 4.3.1 Assembly of $\mathbf{M}_\varphi$

From transformation rule (29a), we have for the block diagonal mass matrix $\mathbf{M}_{\varphi,T_k} \in \mathbb{R}^{N\times N}$ on an arbitrary triangle and the mass matrix $\hat{\mathbf{M}}_\varphi \in \mathbb{R}^{N\times N}$ on the reference triangle $\hat{T}$:

$$\mathbf{M}_{\varphi,T_k} = 2\,|T_k|\,\hat{\mathbf{M}}_\varphi \quad \text{with} \quad \hat{\mathbf{M}}_\varphi \coloneqq \int_{\hat{T}} \begin{bmatrix} \hat{\varphi}_1\hat{\varphi}_1 & \cdots & \hat{\varphi}_1\hat{\varphi}_N \\ \vdots & \ddots & \vdots \\ \hat{\varphi}_N\hat{\varphi}_1 & \cdots & \hat{\varphi}_N\hat{\varphi}_N \end{bmatrix} \mathrm{d}\hat{\mathbf{x}}\,.$$

Then the global mass matrix $\mathbf{M}_\varphi$ can be easily computed by

$$\mathbf{M}_\varphi = \begin{bmatrix} \mathbf{M}_{\varphi,T_1} & & \\ & \ddots & \\ & & \mathbf{M}_{\varphi,T_K} \end{bmatrix} = 2 \begin{bmatrix} |T_1| & & \\ & \ddots & \\ & & |T_K| \end{bmatrix} \otimes \hat{\mathbf{M}}_\varphi\,,$$

where we make use of the Kronecker product. The matrix is assembled in the function `assembleMatElemPhiPhi`.

### 4.3.2 Assembly of $\mathbf{G}^m$

For the assembly of matrices $\mathbf{G}^m$ from (13), we make use of the transformation rule for the gradient (28). 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 (28), (29a) and obtain

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

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

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

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

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

This procedure follows the assembly of $\mathbf{G}^m$ in [28] very closely with the difference being the evaluation of the velocity in the quadrature points instead of using a projected DG representation of the velocity. The global matrix is constructed using standard Kronecker (35) and vectorial Kronecker operator (36)

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

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`.

### 4.3.3 Assembly of S and $\mathbf{S}_{\text{out}}$

Matrices $\mathbf{S}$ and $\mathbf{S}_{\text{out}}$ are assembled together, the set of relevant edges is expanded to $\mathcal{E}_{\text{int}} \cup \mathcal{E}_{\text{out}}$. On a relevant edge $E_{kn}$, we transform terms of the form given in Eq. (15a) using transformation rule (32) and approximate the integral by a one-dimensional numerical quadrature rule

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

where $\mathbf{U}_{\boldsymbol{\nu}} \in \mathbb{R}^{K\times 3\times R}$ holds the normal velocity evaluated in each quadrature point, and the subscript $l$ in $\hat{\mathbf{S}} \in \mathbb{R}^{N\times\overline{N}\times 3\times R\times 2}$ covers the two cases of $\hat{\beta}_{kn}$ in (33). This allows us to assemble the global matrix as

$$\mathbf{S} = \sum_{n=1}^{3} \underbrace{\begin{bmatrix} \delta_{E_{1n}=E_1} & \cdots & \delta_{E_{1n}=E_{\overline{K}}} \\ \vdots & \ddots & \vdots \\ \delta_{E_{Kn}=E_1} & \cdots & \delta_{E_{Kn}=E_{\overline{K}}} \end{bmatrix}}_{=:\boldsymbol{\Delta}_n} \otimes_{\mathrm{V}} \left( \sum_{r=1}^{R}\sum_{l=1}^{2} \begin{bmatrix} \delta_{E_{1n}\in\mathcal{E}_{\text{int}}\cup\mathcal{E}_{\text{out}}}\,|E_{1n}|\,\delta_{\kappa(\rho(1,n),l)=1}\,[\mathbf{U}_{\boldsymbol{\nu}}]_{1,n,r} \\ \vdots \\ \delta_{E_{Kn}\in\mathcal{E}_{\text{int}}\cup\mathcal{E}_{\text{out}}}\,|E_{Kn}|\,\delta_{\kappa(\rho(K,n),l)=K}\,[\mathbf{U}_{\boldsymbol{\nu}}]_{1,n,r} \end{bmatrix} \otimes [\hat{\mathbf{S}}]_{:,:,n,r,l} \right). \tag{37}$$

Here we introduce the permutation matrix $\boldsymbol{\Delta}_n \in \mathbb{R}^{K\times\overline{K}}$, $n\in\{1,2,3\}$ 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`.

#### 4.3.4 Assembly of $\mathsf{R}_\mu$, $\mathsf{T}$, and $\mathsf{K}_{\mu,\text{out}}$

Matrices $\mathsf{R}_\mu$, $\mathsf{T}$, and $\mathsf{K}_{\mu,\text{out}}$ are all constructed using similar terms with the only difference being the set of edges considered or the roles of $\varphi$ and $\mu$ interchanged. The terms in Eqs. (16a), (21a), or (23b), respectively, are each transformed using transformation rule (32) yielding

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

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

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

with $\boldsymbol{\Delta}_n$ from Eq. (37). These matrices are time-independent and thus are assembled only once in `preprocessProblem`, using routine `assembleMatEdgePhiIntMu`.

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

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

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

#### 4.3.5 Assembly of $\mathbf{F}_{\varphi,\text{in}}$

We apply transformation rule (29b) and numerical quadrature to terms with Dirichlet boundary conditions in the first equation (10a) yielding

$$\begin{aligned}
\int_{E_{kn}} (\mathbf{u}\cdot\boldsymbol{\nu}_{kn})\, c_\mathrm{D}\, \varphi_{ki}\mathrm{d}s &= |E_{kn}| \int_0^1 ((\mathbf{u}(t)\circ\boldsymbol{F}_k\circ\hat{\boldsymbol{\gamma}}_n(s))\cdot\boldsymbol{\nu}_{kn})\; c_\mathrm{D}(t)\circ\boldsymbol{F}_k\circ\hat{\boldsymbol{\gamma}}_n(s)\, \hat{\varphi} i\circ\hat{\boldsymbol{\gamma}}_n(s)\, \mathrm{d}s \\
&\approx |E_{kn}| \sum_{r=1}^{R} \omega_r \underbrace{((\mathbf{u}(t)\circ\boldsymbol{F}_k\circ\hat{\boldsymbol{\gamma}}_n(\hat{q}_r))\cdot\boldsymbol{\nu}_{kn})}_{=:[\mathbf{U}_{\boldsymbol{\nu}}]_{k,n,r}} \underbrace{c_\mathrm{D}(t)\circ\boldsymbol{F}_k\circ\hat{\boldsymbol{\gamma}}_n(\hat{q}_r)}_{=:[\mathbf{C}_\mathrm{D}]_{k,n,r}} \hat{\varphi}_i\circ\hat{\boldsymbol{\gamma}}_n(\hat{q}_r) \\
&= |E_{kn}| \sum_{r=1}^{R} \omega_r\, [\mathbf{U}_{\boldsymbol{\nu}}]_{k,n,r}\, [\mathbf{C}_\mathrm{D}]_{k,n,r}\, \hat{\varphi}_i\circ\hat{\boldsymbol{\gamma}}_n(\hat{q}_r) =: [\mathbf{F}_{\varphi,\text{in}}]_{(k-1)N+i}\,.
\end{aligned}$$

The vector is assembled in routine `assembleVecEdgePhiIntVal` to which we pass the point-wise product of $\mathbf{U}_{\boldsymbol{\nu}}$ and $\mathbf{C}_\mathrm{D}$.

#### 4.3.6 Assembly of $\overline{\mathsf{M}}_\mu$ and $\tilde{\mathsf{M}}_\mu$

For the hybrid mass matrices $\overline{\mathsf{M}}_\mu$ and $\tilde{\mathsf{M}}_\mu$ (cf. Eq. (20) and (22), respectively), we apply transformation rule (32) and obtain

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

With the help of permutation matrices $\mathbf{\Delta}_n$ (cf. Eq. (37)), we obtain the global matrices

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

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

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

#### 4.3.7 Assembly of $\mathbf{K}_{\mu,\mathrm{in}}$

Last, we consider the Dirichlet boundary contributions on inflow boundary edges in the "hybridized" equation. We transform the term in Eq. (4.3.7) as before and approximate it using numerical quadrature

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

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. (33)) is relevant here. Thus the global vector is assembled as

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

which is implemented in `assembleVecEdgeMuFuncCont`.

# 5 Numerical results

In this section, we verify our code by means of convergence experiments followed by some performance analysis of the code concluding with a comparison of the presented HDG discretization to an implicit version of the DG discretization from the previous publication [28].

## 5.1 Analytical convergence tests

Our implementation is verified by showing that the experimental orders of convergence are in agreement with the analytically predicted ones for smooth solutions. For that, we choose an exact solution $c(t,\mathbf{x})$ and velocity field $\mathbf{u}(t,\mathbf{x})$, with which we derive boundary data $c_{\mathrm{D}}$ and source term $h$ analytically by inserting $c$ and $\mathbf{u}$ into (1)–(2). At the end, we compute the discretization error $\|c_h - c\|_{L^2(\Omega)}$ as the $L^2$-norm of the difference between the numerical and the analytical solution as described in our first paper [27]. From that we obtain the experimental orders of convergence EOC as

$$\mathrm{EOC} \coloneqq \ln\left(\frac{\|c_{h_{j-1}} - c\|_{L^2(\Omega)}}{\|c_{h_j} - c\|_{L^2(\Omega)}}\right) \Bigg/ \ln\left(\frac{h_{j-1}}{h_j}\right).$$

| $p$ | 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 convergences 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.

### 5.1.1 Steady problem

To verify our spatial discretization we choose the exact solution $c(\mathbf{x}) \coloneqq \cos(7x^1)\cos(7x^2)$ and velocity field $\mathbf{u}(\mathbf{x}) \coloneqq [\exp((x^1 + x^2)/2), \exp(x^1 - x^2)/2)]^{\mathrm{T}}$ 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 $\mathrm{EOC} = p + 1$ for $p > 0$ as depicted 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 $t_{\mathrm{end}} = 2$ and experimental orders of convergences for different polynomial degrees. We have $h_j = \frac{1}{3 \cdot 2^4}$, $K = 4608$, 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 $h_j = \frac{1}{3 \cdot 2^4}$, 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 $\min(p + 1, 4)$, and Table 3 shows the expected orders of convergence $\mathrm{EOC} = p + 1$ for $p < 4$.

### 5.1.3 Unsteady problem (PDE)

The complete solver is verified using exact solution $c(t, \mathbf{x}) = \cos(7x^1)\cos(7x^2) + \exp(-t)$ and velocity field $\mathbf{u}(\mathbf{x}) \coloneqq [\exp((x^1 + x^2)/2), \exp(x^1 - x^2)/2)]^{\mathrm{T}}$ on $\Omega = (0,1)^2$ in the time interval $J = [0, 2]$. Mesh size and time step are refined according to $h_j = \frac{1}{3 \cdot 2^j}$ and $\Delta t_j = \frac{1}{5 \cdot 2^j}$, respectively. The order of the DIRK scheme is chosen to be $\min(p + 1, 4)$, and Table 4 shows the expected orders of convergence $\mathrm{EOC} = p + 1$ for $0 < p < 4$.

| $p$ | 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 | 6.24e-06 | 4.91 |
| 3 | 1.18e-01 | 0.66 | 4.32e-03 | 2.02 | 8.55e-05 | 3.12 | 6.26e-06 | 3.97 | 2.58e-07 | 4.59 |
| 4 | 6.86e-02 | 0.78 | 1.07e-03 | 2.01 | 1.04e-05 | 3.03 | 3.96e-07 | 3.98 | 3.03e-08 | 3.09 |
| 5 | 3.78e-02 | 0.86 | 2.68e-04 | 2.00 | 1.30e-06 | 3.01 | 2.52e-08 | 3.97 | 5.22e-09 | 2.54 |
| 6 | 2.00e-02 | 0.92 | 6.71e-05 | 2.00 | 1.62e-07 | 3.00 | 1.76e-09 | 3.84 | 9.20e-10 | 2.51 |

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 convergences 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.

## 5.2 Comparison to the DG implementation

Our previous work in series was concerned with the same model problem (1) and flux (2), but a *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 created a variant of the DG solver that incorporates the DIRK schemes and computes the solid body rotation benchmark proposed by LeVeque [38] which we used before to investigate the performance of slope limiters [28].

It consists of a slotted cylinder, a sharp cone, and a smooth hump (see Fig. 1a) placed in a square domain $\Omega = (0,1)^2$ with velocity field $\mathbf{u}(\mathbf{x}) = [0.5 - x^2, x^1 - 0.5]^{\mathrm{T}}$ producing a full counterclockwise rotation of the initial scene over time interval $J = (0, 2\pi)$. With $r = 0.0225$ and $G(\mathbf{x}, \mathbf{x}_0) \coloneqq \frac{1}{0.15}\|\mathbf{x} - \mathbf{x}_0\|_2$, we choose initial data satisfying

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

and homogeneous Dirichlet boundary $c_{\mathrm{D}} = 0$ conditions and right-hand side function $f = 0$. We use an unstructured mesh generated by MATLAB's `initmesh` with maximum element size $h = 2^{-6}$ (resulting in 14006 elements) and 320 time steps.

Figures 1 and 2 show the computed solution at end time $t_{\text{end}} = 2\pi$ for different polynomial degrees. We chose a color map (inspired by [39]) 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 reduced the opacity from above to zero. Figure 3 presents intersection lines for $p = 3$.

Clearly, the lowest-order approximation is unusable for this kind of problem in both implementations with numerical diffusion removing most (DG) or all (HDG) characteristics 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. 3 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 [28], where

| | $\|c_h(t_{\text{end}}) - c^0\|_{L^2(\Omega)}$ | | runtime $[s]$ | |
|---|---|---|---|---|
| | DG | HDG | DG | HDG |
| 0 | 1.87e-01 | 2.35e-01 | 27.1 | 39.8 |
| 1 | 7.25e-02 | 7.78e-02 | 157 | 218 |
| 2 | 5.53e-02 | 5.44e-02 | 797 | 717 |
| 3 | 4.02e-02 | 4.13e-02 | 3980 | 3166 |
| 4 | 4.16e-02 | 4.18e-02 | 10996 | 7199 |

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 (on an Intel Core-i7 4790 CPU (Haswell) with 32 GBytes RAM and MATLAB R2017a).

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 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

| | $h_{\max} = 2^{-4}$ $(K = 872)$ | | | | | $h_{\max} = 2^{-6}$ $(K = 14006)$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | runtime | assembly step | | solver step | | runtime | assembly step | | solver step | |
| $p$ | | $\mathsf{G}$ | $\mathsf{S}$ | $\mathsf{L}^{-1}$ | $\lambda$ | | $\mathsf{G}$ | $\mathsf{S}$ | $\mathsf{L}^{-1}$ | $\lambda$ |
| 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. Percentage numbers for assembly and solvers are relative to the runtime of the time stepping loop, percentage numbers for the assembly of $\mathsf{G}$ and $\mathsf{S}$ (cf. secs. 4.3.3 and 4.3.2) are relative to assembly runtime, and percentage numbers for inversion of $\mathsf{L}$ (cf. sec. 3.4) and solving (26) for $\lambda$ are relative to solver step runtimes. Measurements were done on an Intel Core-i7 4790 CPU (Haswell) with 32 GBytes RAM and MATLAB R2017a.

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). Thus, 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 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 $\mathbf{G}$ and $\mathbf{S}$, 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 significantly 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. 4.3.2 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 DG solver from our previous publication [28] 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 (26) 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, we list the routines added since the first two papers in series [27, 28] presented in two parts: 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 [30], 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 $\varphi_h$, and `Nmu` is the number of degrees of freedom for the 1D polynomials $\mu_h$. Parameter `qOrd` is the order of the quadrature rule, and `basesOnQuad` is a struct that contains basis functions $\varphi_h$ and $\mu_h$ 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 `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, 'outputBasename', ['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', floor(128 / 2^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 (E0T) or vertices (V0T) that are
% interior or have a certain boundary type.
problemData.g.markE0Tint  = problemData.generateMarkE0Tint(problemData.g);
problemData.g.markE0Tbdr  = problemData.generateMarkE0Tbdr(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 inital 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 $\mathbf{M}_\varphi$ 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.
uNormalQ0E0T = computeFuncContNuOnQuadEdge(problemData.g, u1Cont, u2Cont, problemData.qOrd);
% Determine inflow- and outflow edges
[~, W] = quadRule1D(problemData.qOrd);
markE0TbdrOut = problemData.g.markE0Tbdr & sum(bsxfun(@times, reshape(W,1,1,[]), uNormalQ0E0T), 3) > 0;
markE0TbdrIn = problemData.g.markE0Tbdr & ~markE0TbdrOut;
% 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.markE0Tint|markE0TbdrOut,...
                        problemData.hatS, uNormalQ0E0T);
problemData.globKmuOut = assembleMatEdgePhiIntMu(problemData.g, markE0TbdrOut, problemData.hatRmu)';
% Assemble inflow boundary conditions.
cQ0E0T = computeFuncContOnQuadEdge(problemData.g, cDCont, problemData.qOrd);
problemData.globFphiIn = assembleVecEdgePhiIntVal(problemData.g, markE0TbdrIn, ...
                            cQ0E0T .* uNormalQ0E0T, problemData.N, problemData.basesOnQuad);
problemData.globKmuIn = assembleVecEdgeMuFuncCont(problemData.g, markE0TbdrIn, ...
                            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, markE0T, refEdgeMuMu)` assembles mass-matrices $\overline{\mathbf{M}}_\mu$ and $\tilde{\mathbf{M}}_\mu$ for the edge-based basis functions $\mu_h$ as described in Sec. 4.3.6. Array `markE0T` plays the role of the Kronecker delta in the matrix definition by selecting the relevent 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, markE0T, refEdgeMuMu)
Nmu = size(refEdgeMuMu, 1); Kedge = g.numE;
ret = sparse(Kedge * Nmu, Kedge * Nmu);
for n = 1 : 3
```

```
  Kkn = g.areaE0T(:, n) .*  markE0T(:, n) ;
  ret = ret + kron(sparse(g.E0T(:, n), g.E0T(:, n), Kkn, Kedge, Kedge), refEdgeMuMu);
end % for
end % function
```

`ret = assembleMatEdgePhiIntMu(g, markE0T, refEdgePhiIntMu)` assembles matrices $\mathbf{R}_\mu$, $\mathbf{T}$, and $\mathbf{K}_{\mu,\text{out}}$ as detailed in Sec. 4.3.4. As before, `markE0T` 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, markE0T, 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 = markE0T(:,n) .*  g.markSideE0T(:, n, l) .* g.areaE0T(:, n);
    ret = ret + kron(sparse(1 : K, g.E0T(:, n), Rkn, K, Kedge), refEdgePhiIntMu(:, :, n, l));
  end % for
end % for
end % function
```

`ret = assembleMatEdgePhiIntMuVal(g, markE0T, refEdgePhiIntMuPerQuad, valOnQuad)` builds the global matrices $\mathbf{S}$ and $\mathbf{S}_{\text{out}}$ given in Sec. 4.3.3. Both matrices are assembled at once by specifying all interior and outflow edges as `markE0T`. 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 $\mathbf{U}_\nu$ evaluated at the quadrature points of each edge.

```
function ret = assembleMatEdgePhiIntMuVal(g, markE0T, 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);
  markAreaE0T = markE0T(:, n) .* g.areaE0T(:, n);
  for l = 1 : 2
    markAreaSideE0T = markAreaE0T .*  g.markSideE0T(:, n, l);
    for r = 1 : R
      RknTimesVal = RknTimesVal + kron(markAreaSideE0T .* valOnQuad(:, n, r), ...
                    refEdgePhiIntMuPerQuad{l}(:, :, n, r));
    end % for r
  end % for l
  ret = ret + kronVec(sparse(1:K, g.E0T(:, 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. 4.3.2. 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, markE0T, funcCont, basesOnQuad, qOrd)` builds the vector with Dirichlet data $\mathbf{K}_{\mu,\mathrm{in}}$ (cf. Sec. 4.3.7). Inflow edges are specified in `markE0T`, and Dirichlet data is specified as a function handle in `funcCont`.

```
function ret = assembleVecEdgeMuFuncCont(g, markE0T, 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 = markE0T(:, n) .* g.areaE0T(:,n);
  ret(g.E0T(:, n), :) = ret(g.E0T(:, n), :) + (repmat(Kkn, 1, R) .* funcOnQuad) ...
                        * ( repmat(W', 1, Nmu) .* basesOnQuad.mu{qOrd} );
end % for
ret = reshape(ret', [], 1);
end % function
```

`ret = assembleVecEdgePhiIntVal(g, markE0T, valOnQuad, basesOnQuad, qOrd)` assembles the vector with Dirichlet boundary data $\mathbf{F}_{\varphi,\mathrm{in}}$ (cf. Sec. 4.3.5). Once again, `markE0T` indicates inflow edges, and `valOnQuad` provides the boundary flux $(\mathbf{u}\cdot\boldsymbol{\nu})\,c_{\mathrm{D}}$ evaluated at the quadrature points of the edges.

```
function ret = assembleVecEdgePhiIntVal(g, markE0T, valOnQuad, N, basesOnQuad, qOrd)
if nargin < 6,  p = (sqrt(8*N+1)-3)/2; qOrd = 2*p+1; end
[~, W] = quadRule1D(qOrd);
ret = zeros(g.numT, N);
for n = 1 : 3
  Kkn = markE0T(:, n) .* g.areaE0T(:,n);
  for i = 1 : N
    ret(:, i) = ret(:, i) + Kkn .* (squeeze(valOnQuad(:,n,:)) * (W' .* basesOnQuad.phi1D{qOrd}(:,i,n)));
  end % for
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 $\mathbf{L}^{-1}$ 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 = 0;
invA = cell(numBlocks, 1);
for idxBlock = 1 : numBlocks - 1
  idxStart = (idxBlock - 1) * blockSize + 1;
  idxEnd = idxBlock * blockSize;
  invA{idxBlock} = A(idxStart : idxEnd, idxStart : idxEnd) \ speye(blockSize);
end % for
invA{numBlocks} = A(idxEnd + 1 : end, idxEnd + 1 : end) \ speye(n - (numBlocks - 1) * blockSize);
invA = blkdiag(invA{:});
end % function
```

`basesOnQuadEdge = computeBasesOnQuadEdge(N, basesOnQuadEdge, requiredOrders)` evaluates edge basis function $\mu_h$ 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 `markSideE0T` 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 $\kappa$ (cf. eq. (4)) for given element $T_k$ and edge $E_{kn} = E_{\bar{k}}$.

```
function g = computeDerivedGridData(g)
g.markSideE0T = true(g.numT, 3, 2);
for n = 1 : 3
  % Mark element to have local id 1 or 2 at the given edge
  g.markSideE0T(:, n, 1) = g.T0E(g.E0T(:, n), 2) ~= (1:g.numT)';
  g.markSideE0T(:, n, 2) = ~g.markSideE0T(:, 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 $c_{\mathrm{D}}$ before multiplying it with the normal velocity and assembling $\mathbf{F}_{\varphi,\mathrm{in}}$ 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 testcases given in Sec. 5. To shorten the presentation, the actual definitions are omitted here but can be found in GitHub [30].

```
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, 'generateMarkE0Tint', @(g) g.idE0T == 0);
problemData = setdefault(problemData, 'generateMarkE0Tbdr', @(g) g.idE0T ~= 0);
end % function
```

`ret = integrateRefEdgeMuMu(N, basesOnQuadEdge, qOrd)` computes reference blocks $\hat{\mathbf{M}}_\mu$ (see Sec. 4.3.6).

```
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. 4.3.4).

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

`ret = integrateRefEdgePhiIntMuPerQuad(N, basesOnQuad, qOrd)` computes reference blocks $\hat{\mathbf{S}}$ (see Sec. 4.3.3).

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

`ret = integrateRefElemDphiPhiPerQuad(N, basesOnQuad, qOrd)` computes reference blocks $\hat{\mathbf{G}}$ (see Sec. 4.3.2).

```
function ret = integrateRefElemDphiPhiPerQuad(N, basesOnQuad, qOrd)
if nargin < 3, p = (sqrt(8*N+1)-3)/2;  qOrd = max(2*p, 1);  end
[~, ~, W] = quadRule2D(qOrd); 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{qOrd}(:, j) .* basesOnQuad.gradPhi2D{qOrd}(:, i, m);
    end % for
  end % for
end % for
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 = 0.435866521508458999416019451193556842529 2;
    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 fully 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) and confirms the well-known computation 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.

## Acknowledgements

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.

## References

[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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. Arnold, F. Brezzi, Mixed and nonconforming Finite Element methods: Implementation, postprocessing and error estimates, Mathematical Modelling and Numerical Analysis 19 (1985) 7–32.

[12] F. Brezzi, J. Douglas, L. D. Marini, Two families of mixed Finite Elements for second order elliptic problems, Numerische Mathematik 47 (1985) 217–235.

[13] B. Cockburn, J. Gopalakrishnan, A characterization of hybridized mixed methods for second order elliptic problems, SIAM Journal on Numerical Analysis 42 (2004) 283–301.

[14] 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.

[15] 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.

[16] H. Egger, C. Waluga, hp-analysis of a hybrid DG method for Stokes flow, IMA Journal of Numerical Analysis 33 (2) (2013) 687–721.

[17] 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.

[18] N. Nguyen, J. Peraire, B. Cockburn, A hybridizable discontinuous Galerkin method for the incompressible Navier-Stokes equations, AIAA Paper 2010-362.

[19] N. C. Nguyen, J. Peraire, Hybridizable discontinuous Galerkin methods for partial differential equations in continuum mechanics, Journal of Computational Physics 231 (2012) 5955–5988.

[20] 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.

[21] J. Schütz, G. May, A hybrid mixed method for the compressible Navier-Stokes equations, Journal of Computational Physics 240 (2013) 58–75.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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

[30] B. Reuter, F. Frank, FESTUNG: The Finite Element Simulation Toolbox for UNstructured Grids (2017). URL https://github.com/FESTUNG

[31] 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.

[32] V. Aizinger, A geometry independent slope limiter for the discontinuous Galerkin method, in: E. Krause, Y. Shokin, M. Resch, D. Krner, 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.

[33] 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.

[34] R. Alexander, Diagonally implicit Runge-Kutta methods for stiff O.D.E.'s., SIAM Journal of Numerical Analysis 14 (1977) 1006–1021.

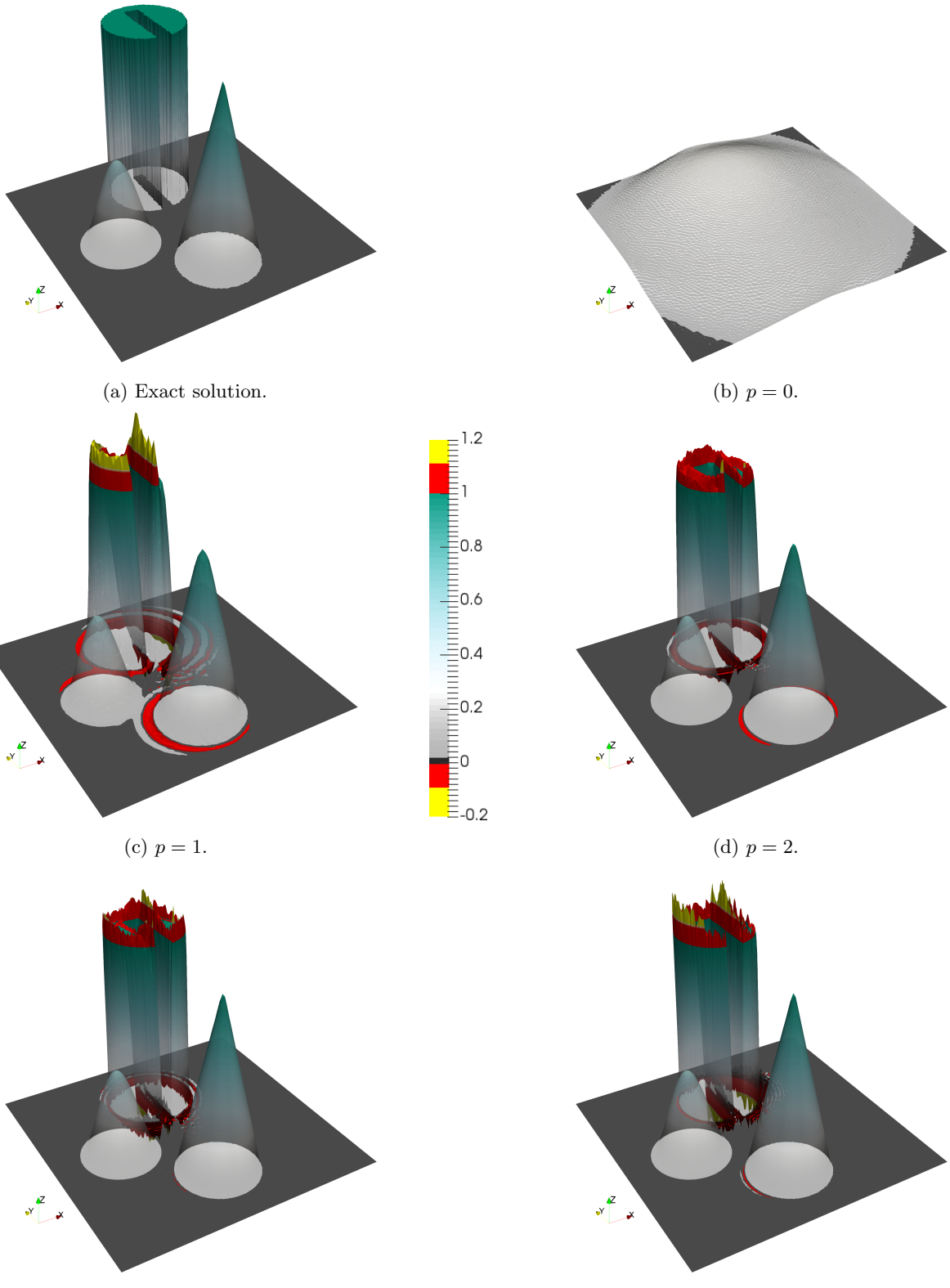
[35] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II, Springer Series in Computational Mathematics, 1991.

[36] 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.

[37] R. Cools, An encyclopaedia of cubature formulas, Journal of Complexity 19 (3) (2003) 445–453. doi:10.1016/S0885-064X(03)00011-6.

[38] 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.

[39] 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) $p = 0$.

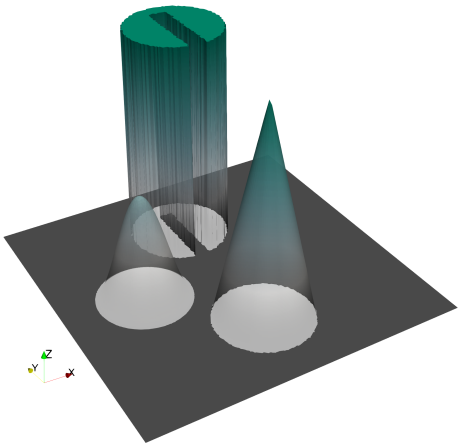
(c) $p = 1$.

(d) $p = 2$.

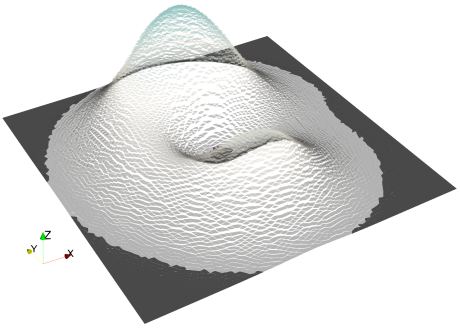
(e) $p = 3$.

(f) $p = 4$.

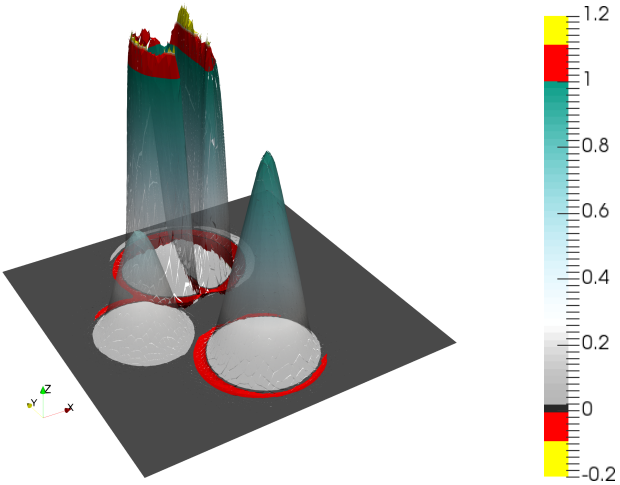
Figure 1: HDG solutions for the solid body rotation benchmark for different polynomial orders at end time $t_{\text{end}} = 2\pi$.



(a) Exact solution.

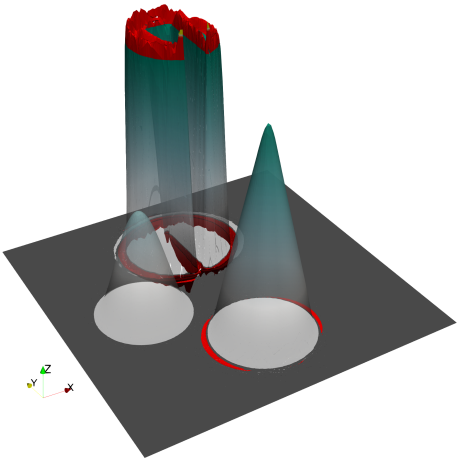


(b) $p = 0$.

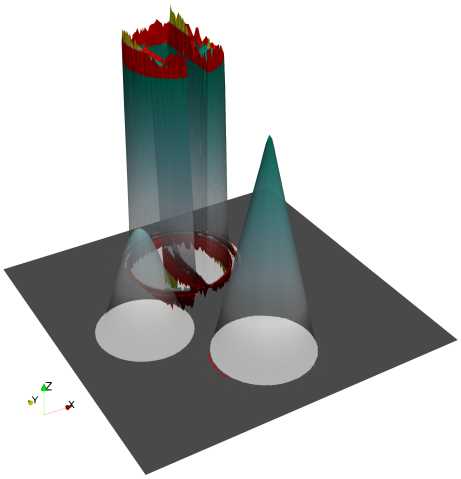




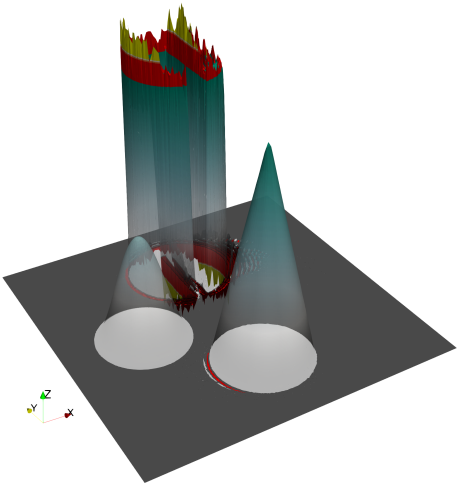

(c) $p = 1$.



(d) $p = 2$.



(e) $p = 3$.



(f) $p = 4$.

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

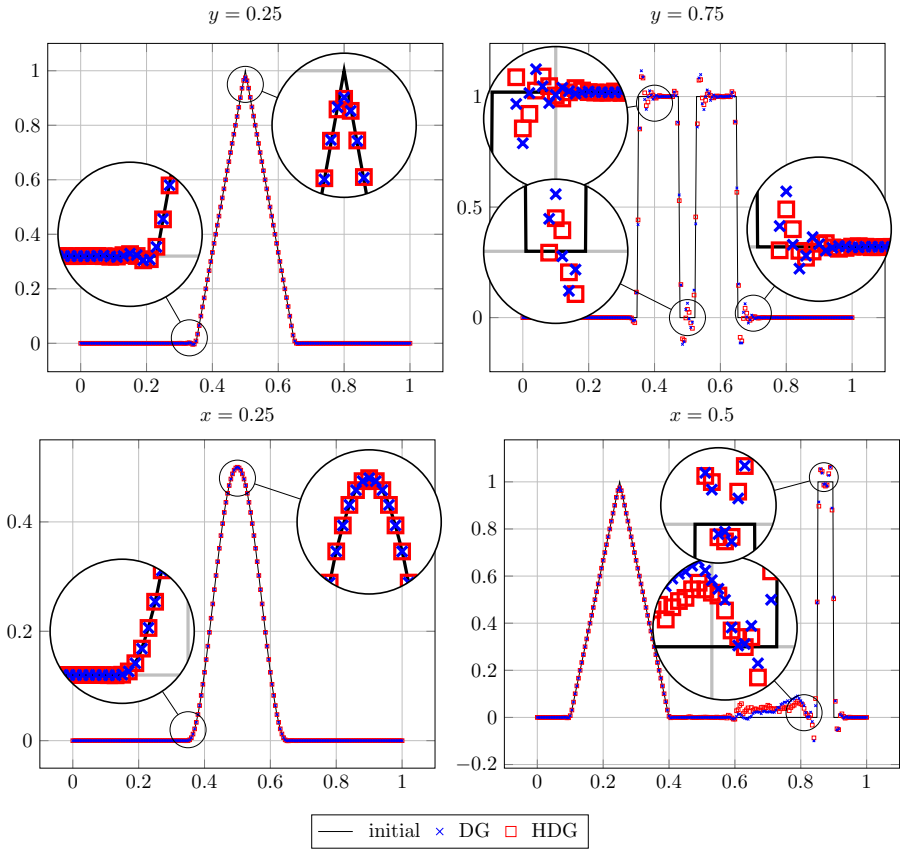



Figure 3: Cross-sections of the DG solutions for the solid body rotation benchmark with $p = 3$ at end time $t_{\text{end}} = 2\pi$.

# UHasselt Computational Mathematics Preprint Series

## 2017

UP-17-09 *Alexander Jaust, Balthasar Reuter, Vadym Aizinger, Jochen Schütz and Peter Knabner*, **FESTUNG: A MATLAB / GNU Octave toolbox for the discontinuous Galerkin method. Part III: Hybridized discontinuous Galerkin (HDG) formulation**, 2017

UP-17-08 *David Seus, Koondanibha Mitra, Iuliu Sorin Pop, Florin Adrian Radu and Christian Rohde*, **A linear domain decomposition method for partially saturated flow in porous media**, 2017

UP-17-07 *Klaus Kaiser and Jochen Schütz*, **Asymptotic Error Analysis of an IMEX Runge-Kutta method**, 2017

UP-17-06 *Hans van Duijn, Koondanibha Mitra and Iuliu Sorin Pop*, **Travelling wave solutions for the Richards equation incorporating non-equilibrium effects in the capillarity pressure**, 2017

UP-17-05 *Hans van Duijn and Koondanibha Mitra*, **Hysteresis and Horizontal Redistribution in Porous Media**, 2017

UP-17-04 *Jonas Zeifang, Klaus Kaiser, Andrea Beck, Jochen Schütz and Claus-Dieter Munz*, **Efficient high-order discontinuous Galerkin computations of low Mach number flows**, 2017

UP-17-03 *Maikel Bosschaert, Sebastiaan Janssens and Yuri Kuznetsov*, **Switching to nonhyperbolic cycles from codim-2 bifurcations of equilibria in DDEs**, 2017

UP-17-02 *Jochen Schütz, David C. Seal and Alexander Jaust*, **Implicit multiderivative collocation solvers for linear partial differential equations with discontinuous Galerkin spatial discretizations**, 2017

UP-17-01 *Alexander Jaust and Jochen Schütz*, **General linear methods for time-dependent PDEs**, 2017

# 2016

UP-16-06 *Klaus Kaiser and Jochen Schütz*, **A high-order method for weakly compressible flows**, 2016

UP-16-05 *Stefan Karpinski, Iuliu Sorin Pop, Florin A. Radu*, **A hierarchical scale separation approach for the hybridized discontinuous Galerkin method**, 2016

UP-16-04 *Florin A. Radu, Kundan Kumar, Jan Martin Nordbotten, Iuliu Sorin Pop*, **Analysis of a linearization scheme for an interior penalty discontinuous Galerkin method for two phase flow in porous media with dynamic capillarity effects** , 2016

UP-16-03 *Sergey Alyaev, Eirik Keilegavlen, Jan Martin Nordbotten, Iuliu Sorin Pop*, **Fractal structures in freezing brine**, 2016

UP-16-02 *Klaus Kaiser, Jochen Schütz, Ruth Schöbel and Sebastian Noelle*, **A new stable splitting for the isentropic Euler equations**, 2016

UP-16-01 *Jochen Schütz and Vadym Aizinger*, **A hierarchical scale separation approach for the hybridized discontinuous Galerkin method**, 2016