A General Object Oriented Framework for Discretizing Nonlinear Evolution Equations

Adrian Burri
Albert-Ludwigs-Universität, Freiburg, Germany
e-mail: burriad@mathematik.uni-freiburg.de

Andreas Dedner
Albert-Ludwigs-Universität, Freiburg, Germany
e-mail: dedner@mathematik.uni-freiburg.de

Dennis Diehl
Albert-Ludwigs-Universität, Freiburg, Germany
e-mail: dennis@mathematik.uni-freiburg.de

Robert Klöfkorn[1]
Albert-Ludwigs-Universität, Freiburg, Germany
e-mail: robertk@mathematik.uni-freiburg.de

Mario Ohlberger[2]
Albert-Ludwigs-Universität, Freiburg, Germany
e-mail: mario@mathematik.uni-freiburg.de

For a large class of non linear evolution problems we derive an abstract formulation that is based on writing the original model as a system of first order partial differential equations. Starting from this reformulation, a set of interface classes are derived that allow a problem independent implementation of various temporal and spacial discretization schemes. In particular, the abstract framework is very well suited for discretizing evolution equations with the Local Discontinuous Galerkin ansatz [1]. The implementation of the proposed framework is done within the Distributed and Unified Numerics Environment **DUNE** [2, 3].

Introduction

In [2, 3] a generic grid interface for serial and parallel computations is introduced that is realized within the Distributed and Unified Numerics Environment **DUNE**. One of the major goals of such an interface based numerics environment is the separation of data structures and algorithms. For instance, the problem implementation can be done on the basis of the interface independent of the data structure that is used for a specific application. Moreover, such a concept allows a reuse of existing codes beyond the interface. This grid interface is a central part

[1]Supported by the Bundesministerium für Bildung und Forschung under contract 03KRNCFR.
[2]Supported by the Landesstiftung Baden-Württemberg under contract 21-665.23/8.

of a number of projects with industrial and physical applications at the Institute for Applied Mathematics in Freiburg. Although a wide range of applications has to be covered, a common ground for the underlying mathematical models can be found. This observation suggests to define also a common interface for the numerical schemes and to provide a generic implementation for the central parts of the schemes. Notwithstanding the diversity of the mathematical models a discretisation using the Local Discontinuous Galerkin ansatz [1] seems very promising. In this paper we describe the strategy for a general implementation of the scheme.

In the next section we outline the Discontinuous Galerkin ansatz for discretizing first order evolution equations and its extension to more complicated evolution equations. To simplify the abstract description the discretization is demonstrated in the case of a simple example. In Section 2 the projects from the Institute of Applied Mathematics in Freiburg which are using the **DUNE** code are presented, focusing on the mathematical models and their formal reformulation so that the local Discontinuous Galerkin ansatz is applicable. Section 3 is devoted to the description of the implementational details of the discretization in the **DUNE** context. We conclude with some results in Section 4.

1. The Discontinuous Galerkin Discretization

1.1. First Order Evolution Equations

We begin our discussion with a study of first order systems of the form: $\partial_t \mathbf{U}(t,\cdot) = \mathcal{L}[\mathbf{U}(t,\cdot)](\cdot)$ where the spatial operator is defined by

$$\mathcal{L}[\mathbf{V}] = \mathbf{S}(\mathbf{V}) - \nabla \cdot \mathbf{F}(\mathbf{V}) - \mathbf{A}(\mathbf{V})\nabla\mathbf{V} \tag{1}$$

where $\mathbf{V} : \mathbb{R}^d \to \mathbb{R}^p$ is in some suitable function space $\mathbb{V}$. We focus here on the space discretization, i.e., we construct a discrete operator $\mathcal{L}_h$ which maps one finite-dimensional function space V_h onto another finite-dimensional function space W_h. This operator is constructed by multiplying equation (1) with a test functions φ and by integrating over the domain $\Omega \subset \mathbb{R}^d$ which is partitioned into a finite number of cells $(\Delta_j)_{j=1,\dots,N}$. We assume that $\mathbf{V}, \varphi$ are smooth functions on the cells Δ_j but may be discontinuous over the cell interfaces. Next, we formally integrate by parts the divergence term over the cells Δ_j and we interpret the nonconservative product $\varphi\mathbf{A}(\mathbf{V})\nabla\mathbf{V}$ as a measure $(\varphi\mathbf{A}(\mathbf{V})\nabla\mathbf{V})_\Phi$ as defined in [4]. Thus we arrive at

$$\begin{aligned}\int_\Omega \mathcal{L}[\mathbf{V}]\varphi d\mathbf{x} \;=\; & \sum_j \int_{\Delta_j} \mathbf{S}(\mathbf{V})\varphi d\mathbf{x} + \\ & \sum_j \int_{\Delta_j} \mathbf{F}(\mathbf{V})\cdot\nabla\varphi d\mathbf{x} - \sum_j \int_{\partial\Delta_j} \widetilde{\varphi\mathbf{F}(\mathbf{V})}\cdot\mathbf{n} d\sigma + \\ & \sum_j \int_{\Delta_j} \varphi\mathbf{A}(\mathbf{V})\nabla\mathbf{V} d\mathbf{x} + \sum_j \int_{\partial\Delta_j} \widetilde{\varphi\mathbf{A}(\mathbf{V})}[\mathbf{V}]\mathbf{n} d\sigma\end{aligned} \tag{2}$$

where $[\mathbf{V}]\mathbf{n}$ denotes the jump of $\mathbf{V}$ in direction of the normal at cell interfaces. $\widetilde{\varphi\mathbf{A}(\mathbf{V})}[\mathbf{V}]\mathbf{n}$ is the value of the measure $(\varphi\mathbf{A}(\mathbf{V})\nabla\mathbf{V})_\Phi$ at a point of discontinuity and the averaged value $\widetilde{\varphi\mathbf{A}(\mathbf{V})}$ depends on the path Φ. The tilde denotes averaging between the cell interfaces, i.e. numerical fluxes for the conservative part. The discrete function $\mathbf{W}_h := \mathcal{L}_h[\mathbf{V}_h]$ is defined so that $\int_\Omega \mathbf{W}_h\varphi_h$ equals the right hand side of (2) for all test functions $\varphi_h \in \mathbb{W}_h$.

The construction of $\mathcal{L}_h$ depends on the averaged values used in (2), which are problem dependent. For the conservative flux average $\widetilde{\mathbf{F}(\mathbf{V})}$ one can either use a generic flux like the Lax-Friedrichs flux function or for example some more sophisticated Riemann solver based flux function [5]. In the simplest case the approximation of the measure consists of the average between the values of $\varphi_h \mathbf{A}(\mathbf{V})$ from both sides of the interface of $\partial\Delta_j$.

1.2. General Systems of Evolution Equations

For the discretization in the case where the spatial operator $\mathcal{L}$ has a more complex form, involving for example higher order derivatives of $\mathbf{V}$ or non-linearities in the non-conservative product, we employ a decomposition of $\mathcal{L}$. Let us assume that we have spatial operators $\mathcal{L}_s$ for $s = 1, \ldots, S$ mapping $\mathbb{V}_{s-1}$ to $\mathbb{V}_s := \mathbb{W}_s \times \mathbb{V}_{s-1}$ where $\mathbb{V}_0 = \mathbb{V}$ and $\mathbb{W}_S = \mathbb{V}$ such that $\mathcal{L} = \Pi_S \circ \mathcal{L}_S \circ \cdots \circ \mathcal{L}_1$ using the projection operator $\Pi_S(V_S, \ldots, V_1, V) := V_S$. If each of the operators $\mathcal{L}_s$ is a first order operator of the form discussed above then we can construct discrete operators $\mathcal{L}_{h,i}$ as before and combine these operators to define $\mathcal{L}_h = \Pi_S \circ \mathcal{L}_{h,S} \circ \cdots \circ \mathcal{L}_{h,1}$.

Let us demonstrate this approach in the case of a simple scalar advection diffusion equation in 1d:

$$\mathcal{L}[u] = -\partial_x\big(f(u) - K(u)\partial_x(k(u))\big) - A(u)\partial_x a(u) \ . \tag{3}$$

This equation differs in two aspects from (1); it contains a second order derivative term and the non-conservative product is non-linear in the derivative term - this leads to problems in defining the measure on the boundary [4]. We therefore formally rewrite this equation in the following form using auxiliary functions $w_1 = (w_{1,1}, w_{1,2})$:

$$\begin{aligned} w_{1,1} &= \partial_x k(u) \ , \\ w_{1,2} &= a(u) \ , \\ w_2 &= -\partial_x(f(u) - K(u)w_{1,1}) - A(u)\partial_x w_{1,2} \ . \end{aligned}$$

Defining the operators $\mathcal{L}_1[u] = (w_1, u)$ and $\mathcal{L}_2[w_1, u] = (w_2, w_1, u)$ we arrive at a decomposition of $\mathcal{L}$ consisting of operators of the desired form.

In some cases closure relations given for example by elliptic operators have to be taken into account. Examples are two-phase flow models and radiation hydrodynamics. In these cases we have additional terms of the form $Q(\mathcal{A}^{-1}(\mathbf{U}))$; here Q is a non-linearity and $\mathcal{A}$ some general operator. These terms can not be directly handled in the form described so far but require the construction of problem dependent discrete operators $\mathcal{A}_h^{-1}$. If these are available they can be directly included in the generic framework described above. In this way the approach can be applied to construct discrete operators for quite complicated systems of evolution equation as we will demonstrate in Section 2.

1.3. Time Discretization

So far we have a semi-implicit discretization of the evolution equation $\partial_t \mathbf{U}(t, \cdot) = \mathcal{L}[\mathbf{U}(t, \cdot)](\cdot)$ which leads to a system of ODEs $\frac{d}{dt}\mathbf{U}_h(t) = \mathcal{L}_h[\mathbf{U}_h(t)]$ for the coefficients defining $\mathbf{U}_h(t)$. To solve this system of ODEs a suitable solver, i.e., a Runge-Kutta method can be applied. Depending on the stability restrictions imposed by the spatial operator one can use either an explicit or an implicit method. To overcome time-step restrictions for explicit schemes while

at the same time retaining the time accuracy of explicit methods for non-restrictive terms a suitable combination of explicit/implicit solvers is sometimes the best approach. To achieve this one has to rewrite the evolution equations using two operators

$$\frac{d}{dt}\mathbf{U}(t,\cdot) = \mathcal{L}_{\text{expl}}[\mathbf{U}(t,\cdot)](\cdot) + \mathcal{L}_{\text{impl}}[\mathbf{U}(t,\cdot)](\cdot) \tag{4}$$

where $\mathcal{L}_{\text{impl}}[\mathbf{U}(t,\cdot)](\cdot)$ combines all the stability restricting terms. The corresponding discrete operators are constructed in the same manner as outlined so far and for the time-discretization a semi-implicit Runge-Kutta method is used — an explicit approach for $\mathcal{L}_{\text{expl,h}}$ and an implicit for $\mathcal{L}_{\text{impl,h}}$.

Again we conclude our discussion with a simple example. Consider the evolution equation where the spatial operator is given by (3). The first order terms lead to a time-step restriction in the order of the grid size h whereas the diffusion term introduces a stability restriction of the order h^2. Therefore an appropriate splitting is given by

$$\begin{aligned} \mathcal{L}_{\text{expl}}[u] &= -\partial_x f(u) - A(u)\partial_x a(u)\ , & (5)\\ \mathcal{L}_{\text{impl}}[u] &= \partial_x(K(u)\partial_x k(u))\ . & (6) \end{aligned}$$

We use a similar decomposition as before, defining $\mathcal{L}_{\text{expl},1}[u] = (w_1, u)$, $\mathcal{L}_{\text{expl},2}[w_1, u] = (w_2, w_1, u)$ and $\mathcal{L}_{\text{impl},1}[u] = (\hat{w}_1, u)$, $\mathcal{L}_{\text{impl},2}[\hat{w}_1, u] = (\hat{w}_2, \hat{w}_1, u)$ by

$$\begin{aligned} w_1 &= a(u)\ ,\\ w_2 &= -\partial_x f(u) - A(u)\partial_x w_1\ ,\\ \hat{w}_1 &= \partial_x k(u)\ ,\\ \hat{w}_2 &= \partial_x(K(u)\hat{w}_1)\ . \end{aligned}$$

This leads to

$$\frac{d}{dt}\mathbf{U}(t,\cdot) = \Pi_2[\mathcal{L}_{\text{expl},2}[\mathcal{L}_{\text{expl},1}[\mathbf{U}(t,\cdot)]]](\cdot) + \Pi_2[\mathcal{L}_{\text{impl},2}[\mathcal{L}_{\text{impl},1}[\mathbf{U}(t,\cdot)]]](\cdot)\ .$$

The simplest time-discretization is given by the forward/backward Euler method. With a time-step Δt this leads to

$$\frac{\mathbf{U}(t^{n+1},\cdot) - \mathbf{U}(t^n,\cdot)}{\Delta t} = \Pi_2[\mathcal{L}_{\text{expl},2}[\mathcal{L}_{\text{expl},1}[\mathbf{U}(t^n,\cdot)]]](\cdot) + \Pi_2[\mathcal{L}_{\text{impl},2}[\mathcal{L}_{\text{impl},1}[\mathbf{U}(t^{n+1},\cdot)]]](\cdot)\ .$$

We conclude that the essential steps for the discretization discussed here are the following

1. rewrite the spatial operator as a sum of two operators $\mathcal{L}_{\text{expl}}, \mathcal{L}_{\text{impl}}$.
2. decompose both spatial operators into first order operators of the form 1.
3. use the Discontinuous Galerkin approach to construct discrete operators.
4. use an explicit/implicit Runge-Kutta ODE solver to advance the solution in time.

Note that steps one and two are problem dependent whereas the following discretization steps three and four can be implemented generically if suitable numerical fluxes are available.

2. Participating Projects

2.1. Torrential Floods

The flow of water-sediment mixtures are often modeled starting from the incompressible two phase 3d Navier-Stokes equations for a non Newtonian fluid with free boundary. To reduce the complexity of the model a shallow flow approach is used which leads to a 2d model where the free boundary is implicitly defined by solving an equation for the depth of the flow h.

The main difficulty involves the derivation of suitable models for the internal and bed friction. Even very simple versions include very complicated terms involving both second order derivatives and non-conservative products. To demonstrate the general ideas for using the approach described previously we only present some part of the model derived in [6]:

$$\partial_t \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) - \mathbf{S}_g(\mathbf{U}) - \mathbf{S}_\tau(\mathbf{U}, \nabla\mathbf{U}, \nabla^2\mathbf{U}) \;=\; 0\ . \tag{7}$$

The balanced quantities are the depth of the flow h and the momentum $h\mathbf{u}$. $\mathbf{S}_g$ is the driving force to gravity, and $\mathbf{S}_\tau$ is a sum consisting of terms modeling the different relevant stresses. We only include two terms in the following

$$\begin{aligned}
\mathbf{U} &= (h, h\mathbf{u})^T\ , \\
\mathbf{F} &= (h\mathbf{u}, h\mathbf{u}\mathbf{u}^T + \frac{1}{2} g_z h^2 \mathcal{I})^T\ , \\
\mathbf{S}_g &= (0, g_z \nabla b h)\ , \\
\mathbf{S}_\tau &= (0, -\mu h \partial_x^2 u_1 + \sin(\phi_{\text{int}}) g_z \text{sign}(\partial_y u_1) h \partial_y h + \ldots, \ldots)\ .
\end{aligned}$$

Here g_z is the magnitude of gravitational acceleration, b describes the bed topology, and ϕ_{int}, μ are constants modeling the internal friction of the granular material.

With a simple transformation we can rewrite equation (7) as a system of first order equations for $\mathbf{U}$ and a set of additional quantities $\mathbf{V}_1$:

$$\mathbf{V}_1 - \nabla u_x \;=\; 0\ , \tag{8}$$

$$\partial_t \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) - \mathbf{S}_g(\mathbf{U}) - \mathbf{S}_2(\mathbf{U}, \mathbf{V}_1) \;=\; 0 \tag{9}$$

where the new source term is given as

$$\mathbf{S}_2(\mathbf{U}, \mathbf{V}_1) \;=\; (0, -\mu h \partial_x V_{1,1} + g_z \sin(\phi_{\text{int}}) \text{sign}(V_{1,2}) h \partial_y h + \ldots, \ldots)\ .$$

Note the non-conservative terms in $\mathbf{S}_2$ which are linear in the derivative as required for the discretization approach described above.

2.2. Two-Phase Flow in Porous Media

In this subsection we consider two-phase multicomponent flow in porous media. Applications in mind are the simulation of water and oil flow through the soil (e.g. biodegradation) or the the simulation of water and gas flow in fuel cells (see for example [7, 8, 9]).

Here, we focus on a simplified system for incompressible two-phase flow including multi-component transport which is the basis for the application in mind. The model considers gas and water flow in a porous medium where in the gas phase also the species hydrogen, oxygen,

water, and some rest gas (mainly consisting of nitrogen) have to be modelled. The governing equations are given as follows. From mass and momentum balance for the water phase ($i = w$) and the gas phase ($i = g$) we get

$$\begin{aligned} \partial_t(\phi s_i) + \nabla \cdot (\mathbf{u}_i) &= q_i(s_i), \\ \mathbf{u}_i &= -\lambda_i(s_i)\mathbf{K}\nabla p_i, \end{aligned} \tag{10}$$

with the following closure relations

$$s_w + s_g = 1, \tag{11}$$

$$p_w = p_g - p_c(s_w) = p_g - p_c(1 - s_g). \tag{12}$$

Within this model the unknown variables are the saturations of the phases s_i, the pressures of the phases p_i, and the phase velocities $\mathbf{u}_i$. ϕ denotes the porosity of the medium and $p_c(s_w)$ is the capillary pressure, given as a function of the water saturation. A parameterization for this function is for example given by the model of Van Genuchten (see [8]). $\lambda_i(s_i)$ denotes the relative mobility of the ith phase, depending on the saturation of the phases, respectively. In addition $\mathbf{K}$ denotes the absolute permeability of the medium and q_i represents sources and sinks. For instance the phase transition between the water and gas phase is modelled through q_i.

As independent variables for our computations we choose the saturation of the gas phase s_g and the pressure of the gas phase p_g. Thereby we assume, that the physics are such that we always have $s_g > 0$. Summing up the equations in (10) and using the relation (11) we get for the so called global velocity $\mathbf{u} = \mathbf{u}_g + \mathbf{u}_w$ and the global source term $q(s) = q_g(s) + q_w(1 - s)$

$$\nabla \cdot \mathbf{u} = q(s_g). \tag{13}$$

The equation for p_g is determined by using relation (12) to replace p_w by $p_g - p_c(1 - s_g)$ and inserting the definitions of $\mathbf{u}, \mathbf{u}_i$ into equation (13). Thus, we get with the elliptic operator $\mathcal{A}(p_g) := -\nabla \cdot ((\lambda_w(1 - s_g) + \lambda_g(s_g))\mathbf{K}\nabla p_g)$:

$$\mathcal{A}(p_g) = q(s_g) - \nabla \cdot (\lambda_w(1 - s_g)\mathbf{K}\nabla p_c(1 - s_g)). \tag{14}$$

For each species in the gaseous phase we have to consider a transport equation. For $\mathbf{c} := (c^{H_2}, c^{O_2}, c^{H_2o}, c^R)^T$ we have

$$\partial_t(\phi s_g \mathbf{c}) + \nabla \cdot (\mathbf{u}_g \mathbf{c}) - \nabla \cdot (\phi s_g \mathbf{D}_g \nabla \mathbf{c}) = \mathbf{r}_g(s_g, s_g\mathbf{c}). \tag{15}$$

The equation for the species R can be dropped by using the relation $c^{H_2} + c^{O_2} + c^{H_2O} + c^R = 1$. Therefore the vector of the concentrations reduces to $\mathbf{c} := (c^{H_2}, c^{O_2}, c^{H_2O})^T$. In this model $\mathbf{D}_g$ denotes the dispersion tensor describing the macroscopic diffusion of the species. $\mathbf{r}_g$ is a source/sink term which for example represents reactions of the species.

Next, we rewrite the above described model as a system of first order equations in order to apply the LDG method. Doing so, we end up with the following set of equations. The vector of unknowns is $\mathbf{U} := (\phi s_g,\, \phi s_g \mathbf{c})^T$ and $\mathbf{V}_1, \mathbf{V}_2$ are vectors of temporary needed values.

$$\begin{aligned} \mathbf{V}_{1,1} + \mathbf{K}\nabla p_c(1 - U_1/\phi) &= 0, \\ \mathbf{V}_{1,2} - \nabla(\mathbf{U}_2/U_1) &= 0, \end{aligned} \tag{16}$$

$$\mathbf{V}_2 + \mathbf{K}\nabla\{\mathcal{A}^{-1}\big(B(\mathbf{V}_{1,1}, U_1/\phi)\big)\} = 0, \tag{17}$$

$$\partial_t \mathbf{U} + \nabla \cdot (\mathbf{F}(\mathbf{U}, \mathbf{V}_1, \mathbf{V}_2)) - \mathbf{S}(\mathbf{U}) = 0. \tag{18}$$

The first pass is formed by equations (16), the second by equations (17); here, p_g is implicitly defined by the $p_g = \mathcal{A}^{-1}(B(\mathbf{V}_{1,1}, V_{1,1}))$ where $B(\mathbf{x}, y) = q(y) - \nabla \cdot (\lambda_w(1-y)\mathbf{x})$ is the right hand side. The last pass consists of equation (18) with the flux function $\mathbf{F}$

$$\mathbf{F}(\mathbf{U}, \mathbf{V}_1, \mathbf{V}_2) := \begin{pmatrix} \lambda_g(U_1/\phi)\mathbf{V}_2 \\ \lambda_g(U_1/\phi)\mathbf{V}_2\mathbf{U}_2/U_1 - U_1\mathbf{D}_g\mathbf{V}_{1,2} \end{pmatrix}$$

end the source/sink terms $\mathbf{S}$

$$\mathbf{S}(\mathbf{U}) := \begin{pmatrix} q_g(U_1/\phi) \\ \mathbf{r}_g(\mathbf{U}/\phi) \end{pmatrix}.$$

2.3. Reactive Navier-Stokes Equations

For the simulation of reactive flows in gas turbine combustors, the reactive Navier-Stokes equations as defined in [10] are used. The mass and momentum conservation part of the compressible Navier-Stokes equations can be written as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \tag{19}$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}^T + p\mathbf{I} - \tau) = 0. \tag{20}$$

In the momentum equation (20), τ denotes the stress tensor which for Newtonian fluids has the form

$$\tau = \mu \left(D\mathbf{u} + (D\mathbf{u})^T - \frac{2}{3}(\nabla \cdot \mathbf{u})\mathbf{I} \right) \tag{21}$$

where μ denotes the viscosity of the fluid and $\mathbf{I}$ the unity tensor.

In order to get to the reactive Navier-Stokes equations, partial differential equation for the chemical species are added:

$$\frac{\partial \rho w_i}{\partial t} + \nabla \cdot (\rho w_i \mathbf{u}) + \nabla \cdot \mathbf{j}_i = \dot{m}_i, \; i \in \{1, n_{species}\} \tag{22}$$

where $\mathbf{j}_i$ is a flux resulting from species diffusion. The source term $\dot{m}_i$ originates from species conversions due to chemical reactions. The species diffusion is modeled using Fick's law $\mathbf{j}_i = -\rho D_i \nabla w_i - \rho \sum_{j=1}^{n_{species}} D_j \nabla w_j$ where D_i is a diffusion constant which can be chosen independently for each species i.

Due to the consideration of diffusion of species with differing enthalpy of formation ΔH_i, the standard energy equation must be supplemented with an additional term, leading to

$$\frac{\partial \rho e}{\partial t} + \nabla \cdot (\mathbf{u}(\rho e + p)) = -\nabla \cdot (\mathbf{j}_T + \tau \mathbf{u} + \sum_{i=1}^{n_{species}} \Delta H_i \mathbf{j}_i). \tag{23}$$

In this equation $\mathbf{j}_T$ is the temperature diffusion flux and is given by Fourier's law as $\mathbf{j}_T = -\lambda \nabla T$ where λ is the heat conduction coefficient and T the temperature.

In order to close the system of equations discussed here, an additional expression relating the thermodynamic variables is needed. Here, the ideal gas law $p = \rho RT$ is used, where the

ideal gas constant of the mixture is expressed as $R = R^* \sum_{j=1}^{n_{species}} \frac{w_j}{W_j}$ with $R^* := 8.136 J/mol$ being the universal gas constant.

If the specific heat capacity c_v is assumed to be independent of the temperature T, the specific total energy ρe is given as

$$\rho e = \frac{1}{\gamma - 1} p + \frac{1}{2}\rho \mathbf{u}^2 + \sum_{i=1}^{n_{species}} \Delta H_i \rho w_i \tag{24}$$

where the definition $\gamma := c_p/c_v$ and the identity $R = c_p - c_v$ is used, as well as the equation of state. Overall we arrive at a set of equations for the partial differential quantities in $\mathbf{U} = (\rho, \rho\mathbf{u}, \rho e, \rho\mathbf{w})^T$ closed using (24) to define the pressure $p = p(\mathbf{U})$ and the relation $T = \frac{p}{R\rho}$. With a simple transformation we finally arrive at a system of first order equations

$$\begin{aligned} \mathbf{V}_1 - \mu\Big(D\mathbf{u} + (D\mathbf{u})^T - \frac{2}{3}(\nabla \cdot \mathbf{u})\mathbf{I}\Big) &= 0, \\ \mathbf{V}_2 + \lambda \nabla \frac{\rho R}{p(\mathbf{U})} &= 0, \\ \mathbf{V}_3 + \rho \mathbf{D} \nabla \mathbf{w} + \rho \sum_{i=1}^{n_{spec}} D_i \nabla w_i &= 0, \end{aligned} \tag{25}$$

$$\partial_t \mathbf{U} + \nabla \cdot \mathbf{F}_2(\mathbf{U}, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3) = \mathbf{S}_2(\mathbf{U}) \tag{26}$$

where the new flux vector $\mathbf{F}_2$ and source term $\mathbf{S}_2$ are given as

$$\begin{aligned} \mathbf{F}_2(\mathbf{U}, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3) &= \Big(\rho\mathbf{u}, \rho\mathbf{u}\mathbf{u}^T + p(\mathbf{U})\mathbf{I} + \mathbf{V}_1, \\ &\qquad (\rho e + p(\mathbf{U}))\mathbf{u} + \mathbf{V}_2 + \mathbf{V}_1\mathbf{u} + \sum_{i=1}^{n_{species}} \Delta H_i \mathbf{V}_{3i}, \rho\mathbf{w}\mathbf{u}^T + \mathbf{V}_3\Big)^T, \\ \mathbf{S}_2(\mathbf{U}) &= \Big(0, 0, 0, \dot{\mathbf{m}}(\mathbf{U})\Big)^T. \end{aligned}$$

2.4. Liquid-Vapour Flows with Phase-Change

We consider the dynamics of a compressible liquid-vapour flow undergoing phase transitions which is governed by the compressible Navier-Stokes-Korteweg system [11]. This is an extension of the Navier-Stokes system with an additional third order term that takes the effect of surface tension at phase boundaries into account.

$$\begin{aligned} \partial_t \rho &+ \nabla \cdot (\rho\mathbf{u}) &&= 0, \\ \partial_t(\rho\mathbf{u}) &+ \nabla \cdot (\rho\mathbf{u}\mathbf{u}^T) + \nabla p(\rho) &&= \nabla \cdot \tau + \lambda\rho\nabla\Delta\rho \end{aligned} \tag{27}$$

where the unknowns are the density $\rho(\mathbf{x}, t) > 0$ and the velocity $\mathbf{u}(\mathbf{x}, t)$, τ is the usual Navier-Stokes tensor as in the section before, the capillarity $\lambda > 0$ is a known constant and the function $p = p(\rho)$ is given by a van-der-Waals equation of state where the temperature is fixed to a constant below the critical temperature to allow for phase transitions. Note that there is no order parameter in this model that indicates the phases. In this model the phases are determined by the value of the density only (low density=vapour, high density=liquid).
The system can also be written in conservative form but it turned out to be better (see [12] and references therein) to discretize the third order term together with the $\nabla p(\rho)$ term in nonconservative form in order to get accurate results when nontrivial local equilibrium configurations

are present.

$$\begin{array}{rcrcl} \partial_t \rho & + & \nabla \cdot (\rho \mathbf{u}) & = & 0, \\ \partial_t (\rho \mathbf{u}) & + & \nabla \cdot (\rho \mathbf{u}\mathbf{u}^T) + \rho \nabla \kappa & = & \nabla \cdot \tau \end{array} \tag{28}$$

where κ is defined by the relation

$$\kappa = \kappa(\rho, \Delta\rho) := -\lambda \Delta \rho + \mu(\rho), \quad \mu(\rho) := \int_0^\rho \frac{p'(s)}{s} ds. \tag{29}$$

The advantage of the nonconservative form is that the variable κ appears explicitly in the equation. κ is equal to some constant at a static equilibrium state (i.e. a state with $\partial_t \rho = 0$ and $\mathbf{u} = \mathbf{0}$). So the discretization of the nonconservative reformulation of the equation leads naturally to a well-balanced scheme.

$$\begin{array}{rcl} \mathbf{V}_\rho - \nabla \rho & = & 0, \\ \mathbf{V}_\mathbf{u} - \nabla \mathbf{u} & = & 0, \end{array} \tag{30}$$

$$\kappa - \mu(\rho) + \lambda \nabla \cdot \mathbf{V}_\rho = 0, \tag{31}$$

$$\begin{array}{rcrcl} \partial_t \rho & + & \nabla \cdot (\rho \mathbf{u}) & = & 0, \\ \partial_t (\rho \mathbf{u}) & + & \nabla \cdot (\rho \mathbf{u}\mathbf{u}^T) + \rho \nabla \kappa - \nabla \cdot \tau(\mathbf{V}_\mathbf{u}) & = & 0. \end{array} \tag{32}$$

3. Implementation of the Discrete Evolution Operator

This section deals with the realization of the mathematical model from section 1 in **DUNE**. Subsection 3.1 describes fundamental concepts of **DUNE** relevant for the LDG ansatz whereas subsections 3.2 and 3.3 explain the newly introduced discretization concepts and their implementation.

3.1. Discrete Functions and Operators

The implementation of a discrete model of a partial differential equation is based on the following general concept of function spaces, functions and operators, that act on functions.

3.1.1. Abstract definition of function spaces and functions

A function space V in our concept is a set of mappings from the domain $D := \mathbb{K}_D^d$ to the range $R := \mathbb{K}_R^n$, e.g.

$$V := \{u : \mathbb{K}_D^d \to \mathbb{K}_R^n\}$$

Here, $\mathbb{K}_D$ denotes the domian field, $\mathbb{K}_R$ the range field and d, n the dimensions of the domain and range, respectively. To further specify the function space, additional properties can be added, e.g. the function are in C^m or do belong to the Sobolev Space H^m.

A discrete function space V_h with finite dimension m is a subset of a function space with the property that the functions are defined locally on the elements e of the underlying computational grid $\mathcal{T}$. If $\hat{e}$ denotes the reference element of e and F_e the mapping $F_e : \hat{e} \to e$, we define the local base function set $V_{\hat{e}}$ on the reference element $\hat{e}$ through

$$V_{\hat{e}} := \{\varphi_1, \cdot, \varphi_{\dim(V_{\hat{e}})}\}.$$

The discrete function space V_h is then given as

$$V_h := \Big\{ u_h \in V : u_h|_e := u_e := \sum_{\varphi \in V_{\hat{e}}} g(u_{e,\varphi})\ \varphi \circ F_e^{-1},\ \text{for all } e \in \mathcal{T} \Big\}.$$

We call $V_e := \text{span}\{\varphi \circ F_e^{-1} : \varphi \in V_{\hat{e}}\}$ a local function space, $u_e \in V_e$ a local function, and $\text{DOF}_e := \{u_{e,\varphi}, \varphi \in V_{\hat{e}}\}$ the set of local degrees of freedom. In oder to incorporate global properties of the discrete function space, the function space has to provide a mapping g between the local degrees of freedom (DOF_e) and the global degrees of freedom $\text{DOF} := \{u_i : i = 0, \cdots, m\}$.

We summarize, that a discrete function space V_h is determined by a function space V, a grid $\mathcal{T}$, the base function sets $V_{\hat{e}}$ for all reference elements $\hat{e}$ and the mapping g from local to global degrees of freedom. A discrete function $u_h \in V_h$ is accordingly defined as a set of local functions u_e where a local function provides access to the local degrees of freedom (DOF_e).

3.1.2. Abstract definition of operators acting on discrete function

A discrete operator L_h is a mapping that acts on discrete functions, e.g.

$$L_h : V_h \to W_h.$$

Thereby, we suppose that a discrete operator may always be decomposed into a global Operator L_{pre}, a set of local operators L_e, and a global operator L_{post}, i.e.

$$\begin{aligned} L_{pre} &: V_h \to \{V_e, e \in \mathcal{T}\}, \\ L_e &: V_e \to W_e, \quad \text{for all } e \in \mathcal{T}, \\ L_{post} &: \{W_e, e \in \mathcal{T}\} \to W_h, \end{aligned}$$

$$L_h = L_{post} \circ \text{diag}\{L_e, e \in \mathcal{T}\} \circ L_{pre}.$$

Here $\text{diag}\{L_e, e \in \mathcal{T}\}$ is a diagonal matrix composed by the entries L_e. Note that with this definition of a discrete operator we are able to combine operators L_h^1 and L_h^2 in a local way, provided that $L_{pre}^2 \circ L_{post}^1 = Id$, i.e.

$$L_h^2 \circ L_h^1 = L_{post}^2 \circ \text{diag}\{L_e^2 \circ L_e^1, e \in \mathcal{T}\} \circ L_{pre}^1.$$

3.1.3. Interface classes for discrete functions and operators

According to the abstract description of discrete functions and operators above, we define the following interface classes:

1. `FunctionSpace⟨DomainField, RangeField, DomainDim, RangeDim⟩`
   This class corresponds to the function space V. It is parameterized by the domain field $\mathbb{K}_D$ =`DomainField`, the range field $\mathbb{K}_R$ =`RangeField`, as well as the dimensions of the domain d =`DomainDim` and range n =`RangeDim`.

2. `Function⟨FunctionSpace⟩`
   A Function is parameterized by the type of the function space `FunctionSpace` it belongs to. To evaluate a function, the following method is provided:

   (a) `evaluate(x,ret)`: Evaluates the function at point `x` and returns the value `ret`.

3. `DiscreteFunctionSpace⟨FunctionSpace, Grid, BaseFunctionSet⟩`
   This class corresponds to the discrete function space V_h. It is parameterized by the type of the function space V =`FunctionSpace` such that $V_h \subset V$, the type of the computational grid $\mathcal{T}$ = `Grid` and the type of the base function set $V_{\hat{e}}$ = `BaseFunctionSet`. The class provides an iterator for the access of the entities e of the grid. In addition the following methods are provided:

   (a) `mapToGlobal(e, nLocal)`: Returns the global number of the degree of freedom with local number `nLocal` on the entity `e`. Thus, it corresponds to the mapping g in our abstract definition.

   (b) `getBaseFunctionSet(e)`: Returns base function set $V_{\hat{e}}$ of entity `e`.

4. `DiscreteFunction⟨DiscreteFunctionSpace, LocalFunction⟩`
   A discrete function is parameterized with the type of the discrete function space V_h = `DiscreteFunctionSpace` it belongs to. In addition it is also parameterized with the type of its local functions u_e `LocalFunction` on the entities e. To access the local functions, the following method is provided:

   (a) `localFunction(e, lf)`: Returns the local function `lf` of entity `e`.

5. `DiscreteOperator⟨LocalOperator,DFDomain,DFRange⟩`
   A discrete operator L_h is parameterized by the type of the functions in its domain (`DFDomain`) and the type of the functions in its range (`DFRange`). In addition the type of the local Operators L_e is given. To apply the discrete operator, the `()`-operator is defined as follows:

   (a) `(arg, dest)`: Applies the operator to `arg` of type `DFDomain` and returns the resulting discrete function `dest` of type `DFRange`.

3.2. Combining Discrete Operators – Abstract Interface Classes

In the following, the classes for combining discrete operators in the form required for the Local Discontinuous Galerkin (LDG) ansatz described in the previous sections are presented.

The new classes in **DUNE** are:

1. `Pass⟨ProblemType, PreviousPass⟩`
   As described in section 1, the operator $\mathcal{L}$ is decomposed into passes $\mathcal{L}_i$ as $\mathcal{L}[\mathbf{U}] = \mathcal{L}_K[\mathcal{L}_{K-1}[\cdots\mathcal{L}_2[\mathcal{L}_1[U]]\cdots]]$. This class serves as the base class for specialized versions of a sub operator $\mathcal{L}_i$. It controls the execution of the previous passes and assembles the data, i. e. it combines the solution of the previous passes $(\mathbf{V}_{i-1}, \ldots, \mathbf{V}_1)$ with the data $\mathbf{U}$. The actual computation of a pass is implemented in a subclass, which needs to override the method `compute` to this end. The class takes two template parameters:

   (a) `ProblemType` User-defined class which provides the problem dependent types.

   (b) `PreviousPass` Type of the previous pass implementing $\mathcal{L}_{i-1}$.
   `PreviousPass` is either a subtype of `Pass` or – in case the actual pass is the first pass – of type `StartPass⟨ArgumentImp⟩`.
   `StartPass⟨ArgumentImp⟩` serves as an end marker to the list of passes.

At run time, the structure of the passes resembles a linked list. Due to the template based implementation, the structure of this linked list can be evaluated at compile-time, so that every pass knows the exact type of its preceding passes and has access to the types defined by them.

2. `PassImpl⟨ProblemType, PreviousPass⟩`
   `PassImpl` stands for a set of generic classes which implement the evaluation of the operator $\mathcal{L}_i$ of pass i with the data obtained from the previous pass. Predefined classes exist for common cases which result from the transformation of the overall problem into a first order system. For instance, specific implementations to evaluate equations of the form (2) with a flux function **F** and a source term **S** and for solving general elliptic problems are available in **DUNE**. If the problem at hand does not fit into one of these categories, the user can define her own classes by deriving from `Pass` and overriding the method `Pass⟨...⟩::compute`.

3. `ProblemImpl`
   `ProblemImpl` stands for a set of user-defined classes which describe the actual problem. In most cases, the class consists of problem-specific type definitions and an implementation for the functions **S**, **F** and **A** in equation (1). Hence, the effort to adapt the Discontinuous Galerkin Operator in **DUNE** is reduced to write a simple problem definition class for each pass while the more intricate details are shielded from the user in the predefined classes.

3.3. Combining Discrete Operators – Example Implementation

To illustrate how the classes described in section 3.2 are used, a simplified implementation is shown here. The code example implements a simplified version of the problem (1) defined in section 1, using a finite difference discretisation for simplicity. The simplified form of the equation reads

$$\partial_t U(x,t) = -\partial_x \cdot (aU(x,t) + \epsilon \partial_x U(x,t)). \tag{33}$$

The core of the implementation is the class `Pass`. In the simplest form, an operator would contain a list of `Pass` pointers, sequentially evaluate them with the data and the results from the previous `Pass`es and return the result from the last one. The solution implemented here is close to this form, with two differences:

1. There is no enclosing operator class, but the last pass serves as the operator itself. Consequently, the assembly of the correct input data to each `Pass` and the adherence to the right calling order is managed by the `Pass` class itself.

2. The list of `Pass`es is implemented as a linked list evaluable at compile-time so that type information can be carried over from previous `Pass`es. This is necessary since the type of the discrete functions of the data and of the results varies from `Pass` to `Pass`.

As an additional responsibility, the `Pass` class needs to provide storage for intermediate results which are only used within the passes. The following listing shows a simplified implementation of this core class.

```
// The Pass base class
template <class Problem, class FunctionSpace, class PreviousPass>
class Pass {
public:
  enum {passnr=PreviousPass::passnr+1};
  typedef FunctionSpace DestinationType;
  // Types from previous pass
  typedef typename PreviousPass::GlobalArgumentType GlobalArgumentType;
  typedef typename PreviousPass::NextArgumentType LocalArgumentType;
  // Types for internal usage or next pass
  typedef Pair<const GlobalArgumentType*, LocalArgumentType>
          TotalArgumentType;
  typedef Pair<DestinationType*, LocalArgumentType> NextArgumentType;
public:
  // The pass i is initialized with its precessor pass i-1. Every pass triggers
  // the allocation of temporary memory on its precessor. In doing this it is
  // guaranteed that the last pass allocates no temporary memory for the result
  // (which is passed to it by the client)
  Pass(PreviousPass& pass) :
    previousPass_(pass), dest_(0) {
    pass.initialize()
  }
  virtual ~Pass() {
    delete dest_;
    dest_ = 0;
  }
  // The application operator is called directly by the client. This way,
  // the last pass serves as the operator in the Dune sense and the memory
  // for the result (the argument dest) is passed to it directly.
  void operator() (const GlobalArgumentType& arg, DestinationType& dest)
  {
    // Evaluate the previous passes first
    previousPass_.pass(arg);
    // Build up the argument list (consisting of the data obtained
    // from the DGOperator and the results of the previous passes)
    TotalArgumentType totalArg(&arg, previousPass_.localArg());
    // Do computations related to this pass (virtual function)
    compute(totalArg, dest);
  }
  // Trigger the allocation of temporary memory
  virtual void initialize() {}
private:
  // The functions here are not part of the public interface but must be
  // accessible by all other passes
  template <class P, class PP, class PPP>
  friend class Pass;
  // Pass is called from preceding pass. The only difference to the
  // application operator is that the use of this function causes the pass
  // to use its own temporary discrete function instead of a given one
  void pass(const GlobalArgumentType& arg) {
    operator()(arg, *dest_);
  }
  // Return a list of the results of all previous passes including this one
  NextArgumentType localArg() {
    return NextArgumentType(dest_, previousPass_.localArg());
  }
private:
  // The actual computations are delegated to a subclass
  virtual void compute(const TotalArgumentType& arg,
                       DestinationType& dest) = 0;
  PreviousPass& previousPass_;
protected:
  // Temporary storage. Subclass must initialism it when needed.
  mutable DestinationType* dest_;
};
```

The implementation of the method `compute` is deferred to a subclass in order to allow for different implementations, tailored to the user's need. Moreover, `Pass` hides the intricate details and compile-time constructs from the user and from the implementor of specific passes. The following listing shows an example of such a derived `Pass` class. The `FDPass` class implements

the finite difference evaluation of a numerical flux `g`, defined by its `Problem` template argument. The `compute` method consists of an iteration over the grid which calls the numerical flux on each grid node and returns the resulting update vector to the caller. Note that `FDPass` itself is a generic implementation that works for a wide range of problem definitions, which the user can express by writing a respective `Problem` class.

```
// Generic implementation of a first order FD scheme
// Problem must provide numeric flux function g
template <class Problem, class FunctionSpace, class PreviousPass>
class FDPass : public Pass<Problem, FunctionSpace, PreviousPass> {
public:
  typedef Pass<Problem, FunctionSpace, PreviousPass> BaseType;
  typedef typename BaseType::TotalArgumentType ArgumentType;
  typedef FunctionSpace DestinationType;
  enum {passnr=Pass<Problem, FunctionSpace, PreviousPass>::passnr};
public:
  // Constructor initializes base class and creates storage
  FDPass(PreviousPass& prevPass,Problem& prob,int pN) :
    problem(prob), BaseType(prevPass), gridsize_(pN) {}
  // Builds up temporary storage for pass (the V_i)
  virtual void initialize() {
    this->dest_=new DestinationType(gridsize_*Problem::range);
  }
private:
  typename Problem::ConsType rflux,w;
  // Compute provides the actual evaluation of the operator belonging to
  // this pass. arg is a list containing the argument data
  // as well as the results
  virtual void compute(const ArgumentType& arg, DestinationType& dest) {
    dest.clear();
    double h=1./double(dest.size());
    StateVector<Problem::domain> xl,xr;
    xl[0]=0.;
    xr[0]=1.;
    w.clear();
    for (int i=0;i<dest.size();++i) {
      if (i<dest.size()-1)
        problem.g(i,i+1,xr,xl,arg,rflux);
      else
        rflux.clear();
      w-=rflux;
      w/=(-h);
      dest.set(i,w);
      w=rflux;
    }
  }
  Problem &problem;
  int gridsize_;
};
```

The next listing shows how the problem can be put into code. The class `Problem` implements the scalar linear transport problem with an additional diffusion term from equation (33). Note that this is the only part the user must write. Very little information about the other passes is necessary, namely which components of the argument type contain the relevant information. To simplify the usage, the problem class acts as an operator itself by just forwarding the call to the last pass.

```
// The problem class organizes the passes and provides the functionality of an
// operator which can be used in a time-integrator.
class Problem {
 private:
  // Problem class for diffusion-pass dx (eps*v)
  class ProblemDiffusion {
  public:
    enum {domain=1};
    enum {range=1};
    typedef StateVector<range> ConsType;
```

```
    typedef DiscreteFunction<domain, range> DestinationType;
    ProblemDiffusion(double eps) :
      koeff(eps) {}
    // This method implements the numerical flux of the diffusion term d_xx u
    template <class ArgumentType>
    void g(int eleft,int eright,
           StateVector<domain> xl,StateVector<domain> xr,
           const ArgumentType& arg, ConsType& dest) {
      ConsType ul;
      Element<0>::get(arg)->evaluate(eleft,xl,ul);
      dest=koeff*ul;
    }
  private:
    double koeff;
  };
  // Problem class for Transport-pass dx (a*u-v)
  class ProblemTransport { // dx (a*u-v)
  public:
    enum {domain=1};
    enum {range=1};
    typedef StateVector<range> ConsType;
    typedef DiscreteFunction<domain, range> DestinationType;
    ProblemTransport(double a) : koeff(a) {}
    // Implementation of the numerical (upwind) flux
    template <class ArgumentType>
    void g(int eleft,int eright,
           StateVector<domain> xl,StateVector<domain> xr,
           const ArgumentType& arg, ConsType& dest) {
      ConsType ul,ur;
      Element<0>::get(arg)->evaluate(eleft,xl,ul);
      Element<0>::get(arg)->evaluate(eright,xr,ur);
      if (koeff>0)
        dest=koeff*ul;
      else
        dest=koeff*ur;

      ConsType vr;
      At<diffpass>::get(arg)->evaluate(eright,xr,vr);
      dest-=vr;
    }
  private:
    double koeff;
  };
public:
  // Argument/Destination type for operator
  typedef DiscreteFunction<1,1> GlobalArgumentType;
  typedef GlobalArgumentType GlobalDestinationType;
  // Constructor.
  // The constructor of the problem initializes all problem and pass objects
  Problem(double a,double eps,int N) :
    N_(N),
    problem1(eps), problem2(a),
    pass1(pass0,problem1,N),
    pass2(pass1,problem2,N) {
  }
  // The problem provides its own application operator and acts as a Dune
  // operator in its own right.
  void operator() (const GlobalArgumentType& arg, GlobalDestinationType& dest) {
    pass2(arg,dest);
  }

  // Size of the grid
  int size() {
    return N_;
  }
private:
  typedef PassStart<GlobalArgumentType> PassStartType;
  typedef FDPass<ProblemDiffusion, DiscreteFunction<1,1>, PassStartType>
    Pass1Type;
  typedef FDPass<ProblemTransport, DiscreteFunction<1,1>, Pass1Type>
    Pass2Type;
```

```
  int N_;
  ProblemDiffusion problem1;
  ProblemTransport problem2;
  PassStartType pass0;
  Pass1Type pass1;
  Pass2Type pass2;
  enum {diffpass=Pass1Type::passnr,transpass=Pass2Type::passnr};
};
```

Putting it all together consists of instantiating the problem class and initializing the correct data. The evaluation of the operator finally is as simple as writing `prob(data, sol)`, as can be seen in the next listing. Here, the information about the time-stepping scheme is handled in an additional class (`ForwardEuler`) not presented in this paper.

```
int main(int argc, char ** argv) {
  int N=atoi(argv[1]); // number of grid nodes
  double T=atof(argv[2]); // end time
  double eps=atof(argv[3]); // viscosity
  double a=atof(argv[4]); // advection velocity
  // Creation of the problem and the corresponding operator
  Problem prob(a,eps,N);
  // Creation of the data vectors
  Problem::GlobalArgumentType u0(N), u(N);
  // Creation of the timestepping operator (not defined here)
  ForwardEuler<Problem> rk(prob,0.);
  // Initialization of the data
  initData(u, u0);
  //Time loop
  while (rk.time()<T) {
    rk(u,u);
    std::cerr << rk.time() << std::endl;
  }
  // Output result
  outputResult(u);
}
```

4. Results and Conclusion

We apply the method to the Navier-Stokes-Korteweg system described in Subsection 2.4. For definition of the averaged values and numerical fluxes we refer to [12]. Time discretization is done by application of implicit Runge-Kutta methods.

As test case for the method we choose an initial configuration in two space dimensions for which the solution of the Navier-Stokes-Korteweg system is *quasi*-known. This means the solution is known to exist and can be computed very accurately by a different scheme. Figure 1 shows the result of this computation. The left part of the figure shows that the expected order of the method will be reached, the right figure demonstrates the efficiency of the method i.e. higher order methods lead to more efficient methods (provided that the solution is smooth).

References

[1] B. Cockburn, C.-W. Shu (1998) The local discontinuous Galerkin method for time-dependent convection-diffusion systems. SIAM J. Numer. Anal., 35: 2440–2463.

[2] P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, M. Rumpf (2004) Towards a Unified Framework for Scientific Computing. In Proc. of the 15th International Conference on Domain Decomposition Method.

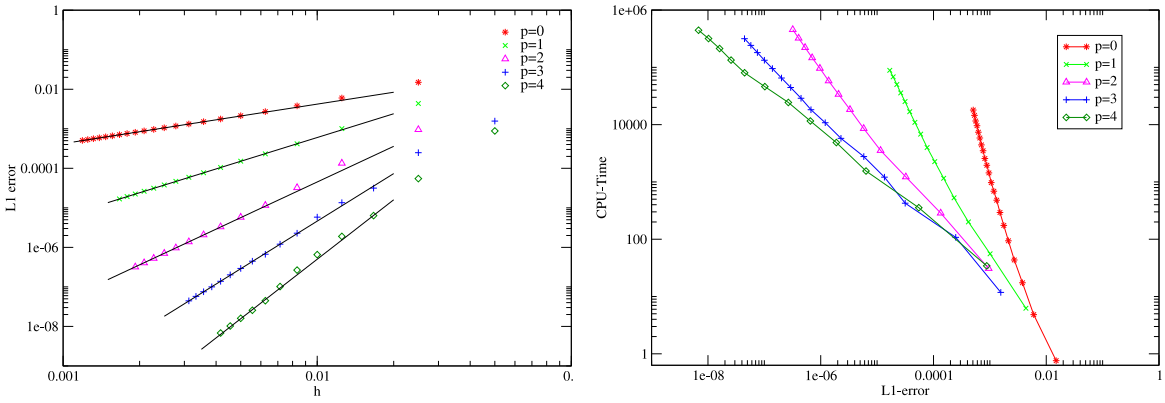

Fig. 1. DG approximation for ansatz functions with different polynomial degree p. Mesh size h versus L^1-error (the black lines indicate the expected order) and L^1-error versus CPU-time.

[3] A. Burri, A. Dedner, R. Klöfkorn, M. Ohlberger (2005) An efficient implementation of an adaptive and parallel grid in DUNE. Tech. rep., Submitted to: Proceedings of The 2nd Russian-German Advanced Research Workshop on Computational Science and High Performance Computing, Stuttgart, March 14 - 16.

[4] G. Dal Maso, P. LeFloch, F. Murat (1995) Definition and weak stability of nonconservative products. J. Math. Pures Appl., 74: 483–548.

[5] R. LeVeque (1990) Numerical Methods for Conservation Laws. Lectures in Mathematics, Birkhäuser, first edn.

[6] P. Vollmöller (2004) A shock capturing wave propagation method for dry and fluid-saturated granular flows. J. Comput. Phys., 199 (1): 150–174.

[7] P. Bastian, B. Rivière (2004) Discontinuous galerkin methods for two-phase flow in porous media. Tech. Rep. 2004–28, IWR (SFB 359), Universität Heidelberg.

[8] R. Helmig (1997) Multiphase Flow and Transport Processes in the Subsurface: A contribution to the modeling of hydrosystems. Springer.

[9] K. Kühn, M. Ohlberger, J. Schumacher, R. Ziegler, R. Klöfkorn (2003) A dynamic two-phase flow model of proton exchange membrane fuel cells. Preprint CSCAMM Report 03-07, Submitted to the 2nd EUROPEAN PEFC FORUM, Luzern.

[10] T. Poinsot, D. Veynante (2001) Theoretical and Numerical Combustion. R.T. Edwards, Inc.

[11] D. Anderson, G. McFadden, A. Wheeler (1998) Diffuse interface methods in fluid mechanics. Ann. Rev. Fluid Mech., 30: 139–165.

[12] D. Diehl (2005) Well balanced discontinuous Galerkin schemes for the Navier-Stokes-Korteweg equations. Proceedings of HYP 2004, Yokohama Publishers, Inc.