Archive of Numerical Software
vol. 5, no. 1, 2017, pages 21–62
DOI: 10.11588/ans.2017.1.28602

The DUNE-FEM-DG module

Andreas Dedner[1], Stefan Girke[2], Robert Klöfkorn[3], and Tobias Malkmus[4]

[1]University of Warwick, UK
[2]University of Münster, Germany
[3]International Research Institute of Stavanger, Norway
[4]University of Freiburg, Germany

Received: March 3rd, 2016; **final revision:** November 17th, 2016; **published:** March 6th, 2017.

Abstract: In this paper we discuss the new publicly released DUNE-FEM-DG module. This module provides highly efficient implementations of the Discontinuous Galerkin (DG) method for solving a wide range of non linear partial differential equations (PDE). The interfaces used are highly flexible and customizable, providing for example mechanisms for using distributed parallelization, local grid adaptivity with dynamic load balancing, and check pointing. We discuss methods for solving stationary problems as well as a matrix-free implementation for time dependent problems. Both parabolic and first order hyperbolic PDE are discussed in detail including models for compressible and incompressible flows, i.e., the Navier-Stokes equations.

For the spatial discretization a wide range of DG methods are implemented, ranging from the standard interior penalty method to methods like LDG and CDG2. Upwinding numerical fluxes for first order terms are also available, including limiter based stabilization for convection dominated PDEs. For the temporal discretization Runge-Kutta methods are used, including higher order explicit, diagonally implicit and IMEX schemes. We discuss asynchronous communication, shared memory parallelization, and automated code generation which combined result in a high floating point performance of the code.

Keywords. Numerical Software, DUNE, Discontinuous Galerkin Schemes, Hyperbolic problems, Elliptic problems, Parabolic problems, Euler, Navier-Stokes, Advection-Diffusion, Stokes, Poisson

1 Introduction

In this paper we introduce the DUNE-FEM-DG module which has been recently published under the GNU General Public License version 2, or (at your option) any later version.

The DUNE-FEM-DG module is based on DUNE-FEM (see Dedner et al. [2010b]) and makes use of the infrastructure implemented by DUNE-FEM, e.g. `DiscreteFunctionSpace`, `DiscreteFunction`, and `LocalFunction` or `DofManager`, `AdaptationManager` and `CommunicationManager` for seamless integration of parallel-adaptive Finite Element based discretization methods. A similar approach is provided by the DUNE module DUNE-PDELAB described in Bastian et al. [2010]. Similarities can be found for example in concepts like `DiscreteFunctionSpace` (DUNE-FEM) and `GridFunctionSpace` (DUNE-PDELAB). Differences, on the other hand, exist in the approach to

create discrete function spaces for coupled or vector valued problems, callback (DUNE-FEM) vs tree based approach (DUNE-PDELAB) or the handling of adaptive problems. Here, DUNE-FEM introduces the `AdaptivePersistentIndexSet`, an extension of the `IndexSet` approach from DUNE-GRID, which has been proven to be a more efficient approach (c.f Klöfkorn [2009], Klöfkorn and Nolte [2012]) in comparison to the approaches offered by the DUNE grid interface which are used by DUNE-PDELAB. The most prominent difference to DUNE-PDELAB is the fact, that DUNE-FEM itself does not implement any discretization schemes. This is accomplished by sub modules such as DUNE-FEM-DG or DUNE-ACFEM[1]. DUNE-FEM-DG focuses exclusively on Discontinuous Galerkin (DG) methods for various types of problems. The discretizations used in this module are described by two main papers, Dedner and Klöfkorn [2011] where we introduced a generic stabilization for convection dominated problems that works on generally unstructured and non-conforming grids and Brdar et al. [2012a] where we introduced a parameter independent DG flux discretization for diffusive operators.

Besides the two DUNE related packages mentioned above DG methods have been studied intensively by many other groups and many software packages exist. However, most of these packages do not combine the following features: unstructured grids for 2, and 3 space dimensions, grid adaptivity, parallel computing capabilities, and open-source licenses. Combining all these constraints there are only a few packages available, for example, deal.II (Bangerth et al. [2007]), feel++ (The Feel++ Consortium [2015]), or Nektar++ (Karniadakis and Sherwin [2005]).

Compared to DUNE-PDELAB which has strong support for incompressible problems, DUNE-FEM-DG concentrates on the implementation of matrix-free approaches mainly used for compressible problems but also offers more variety of DG methods. A comparison between DG methods implemented in DUNE-FEM-DG and DUNE-PDELAB for Poisson type problems is given in Eymard et al. [2011]. However, this comparison only compares the quality and effectiveness of the DG methods but not the implementation itself. A fair comparison of both packages with respect to implementation should be subject of a separate study.

Over time several state of the art techniques such as *communication hiding*, *automated code generation*, and *hybrid parallelization* have been added. Consequently, the module has been used in several applications. Most notably a comparison with the production code of the German Weather Service COSMO has been carried out for test cases for atmospheric flow (Brdar et al. [2013], Schuster et al. [2014]). In addition several PhD theses have been conducted (Klöfkorn [2009], Brdar [2012]) or are currently going on (Girke [2017], Malkmus [2017]). The following research publications (journal and conference papers) discuss or make use of DUNE-FEM-DG:

- DG discretizations (Klöfkorn [2009], Dedner and Klöfkorn [2011], Brdar et al. [2012a], Brdar et al. [2012b], Dedner and Klöfkorn [2009], Burri et al. [2006], Brdar [2012])
- Elliptic problems as part of the FVCA V and IV benchmarks on anisotropic diffusion problems (Dedner and Klöfkorn [2008], Klöfkorn [2011], Eymard et al. [2011], Dedner et al. [2014])
- Model reduction (Dedner et al. [2013], Dedner et al. [2011a])
- Meteorological problems (Brdar et al. [2013], Schuster et al. [2014], Dedner and Klöfkorn [2016], Klöfkorn [2012], Brdar et al. [2011b], Brdar et al. [2011a])
- Free surface shallow water flow (Dedner et al. [2011b])
- Atherosclerotic plaque simulation (Girke et al. [2014])
- Reactive flow in moving domains (Klöfkorn and Nolte [2014])
- Hyperbolic systems (Dedner et al. [2010a], Dedner and Giesselmann [2015])

[1] https://gitlab.dune-project.org/dune-fem/dune-acfem.git

A strength of the Dune-Fem-DG module is the general application area, i.e. convection dominated as well as diffusion dominated problems, for 1d, 2d, and 3d models, including parallelization and local grid adaptivity. The model includes the implementation of several discretizations for the second order terms such as Interior Penalty (and variants), Compact Discontinuous Galerkin 1 and 2, and Bassi-Rebay 1 and 2 as well as Local Discontinuous Galerkin. For the first order terms different numerical flux functions can be used and limiter based stabilization is available. For the time discretization a method of lines approach is adopted and a number of implicit, explicit, and IMEX schemes are available. Overall the module thus offers not only a strong base for building state of the art simulation tools but it also allows for method comparison and comparative studies. Adaptivity, including h-p adaptivity, is included and can be used seamlessly. Recent development has been focused on applications with moving grids and multi model applications.

So far, the Dune-Fem-DG module has not been publicly available nor a detailed software description has been available. This is provided, for the first time, by this paper including a revision to allow for easy setup of coupled multi-physics problems. The paper is organized as follows. In Section 2 we describe the DG discretizations. In Section 3 we introduce the main interface classes to implement a mathematical model and in Section 4 we present numerical examples for all the different types of problems covered by Dune-Fem-DG.

2 Governing Equations

In this paper we consider a general class of stationary and time dependent advection-diffusion-reaction problems for a vector valued function $\boldsymbol{U}\colon (0,T)\times\Omega \to \mathbb{R}^r$ with $r \in \mathbb{N}^+$ components. The time dependent problem is a general nonlinear partial differential equation in $\Omega \subset \mathbb{R}^d$, $d = 1,2,3$ of the form

$$\begin{aligned} \partial_t \boldsymbol{U} &= \mathcal{L}(\boldsymbol{U}) && \text{in } (0,T]\times\Omega, \\ \boldsymbol{U}(0,\cdot) &= \boldsymbol{U}_0(\cdot) && \text{in } \Omega, \end{aligned} \tag{1}$$

with

$$\mathcal{L}(\boldsymbol{U}) := -\nabla\cdot\Big(\mathcal{F}(\boldsymbol{U}) - \mathcal{A}(\boldsymbol{U},\nabla\boldsymbol{U})\Big) + \mathcal{S}(\boldsymbol{U}) \tag{2}$$

and suitable boundary conditions. Note that all the coefficients in the partial differential equation are allowed to depend explicitly on the spatial variable x and on time t but to simplify the presentation we suppress this dependency in our notation.

2.1 Spatial Discretization

We first focus on the spatial discretization. In this case our model problem is a stationary system of partial differential equations:

$$\mathcal{L}(\boldsymbol{U}) = 0 \qquad \text{in } \Omega. \tag{3}$$

with either Dirichlet, Neumann, or Robin type boundary conditions. The spatial operator is given by (2).

The considered discretization is based on the Discontinuous Galerkin (DG) approach and implemented in Dune-Fem (Dedner et al. [2010b]) a module of the Dune framework (Bastian et al. [2008a,b]). The discretization is derived in the following way. Given a tessellation $\mathcal{T}_h$ of the domain Ω with $\cup_{K\in\mathcal{T}_h} K = \Omega$ the discrete solution $\boldsymbol{U}_h$ is sought in the piecewise polynomial space

$$V_h = \{\boldsymbol{v} \in L^2(\Omega,\mathbb{R}^r) : \boldsymbol{v}|_K \in [\mathcal{P}_k(K)]^r,\ K \in \mathcal{T}_h\} \quad \text{for some } k \in \mathbb{N},$$

where on simplicial elements $\mathcal{P}_k(K)$ is a space containing polynomials up to degree k while on quadrilateral or hexahedral elements $\mathcal{P}_k(K)$ contains mapped functions given by products of Legendre polynomials of up to degree k in each coordinate on a reference cube. Other basis functions could be chosen without loss of generality.

We denote with Γ_i the set of all intersections between two elements of the grid $\mathcal{T}_h$ and accordingly with Γ the set of all intersections, also with the boundary of the domain Ω. The following discrete form is not the most general but still covers a wide range of well established DG methods. For all basis functions $\boldsymbol{\varphi} \in V_h$ we define

$$\langle \boldsymbol{\varphi}, \mathcal{L}_h(\boldsymbol{U}_h)\rangle := \langle \boldsymbol{\varphi}, \mathcal{K}_h(\boldsymbol{U}_h)\rangle + \langle \boldsymbol{\varphi}, \mathcal{I}_h(\boldsymbol{U}_h)\rangle \tag{4}$$

with the element integrals

$$\langle \boldsymbol{\varphi}, \mathcal{K}_h(\boldsymbol{U}_h)\rangle \quad := \quad \sum_{K\in\mathcal{T}_h} \int_K \Big((\mathcal{F}(\boldsymbol{U}_h) - \mathcal{A}(\boldsymbol{U}_h, \nabla \boldsymbol{U}_h)) : \nabla\boldsymbol{\varphi} + S(\boldsymbol{U}_h)\cdot\boldsymbol{\varphi}\Big), \tag{5}$$

and the surface integrals (by introducing appropriate numerical fluxes $\widehat{\mathcal{F}}_e$, $\widehat{\mathcal{A}}_e$ for the convection and diffusion terms, respectively)

$$\begin{aligned}\langle \boldsymbol{\varphi}, \mathcal{I}_h(\boldsymbol{U}_h)\rangle \quad := \quad & \sum_{e\in\Gamma_i} \int_e \Big(\{\mathcal{A}(\boldsymbol{U}_h, [\![\boldsymbol{U}_h]\!]_e)^T : \nabla\boldsymbol{\varphi}\}_e + \{\mathcal{A}(\boldsymbol{U}_h, \nabla\boldsymbol{U}_h)\}_e : [\![\boldsymbol{\varphi}]\!]_e \Big) \\ & - \sum_{e\in\Gamma} \int_e \Big(\widehat{\mathcal{F}}_e(\boldsymbol{U}_h) - \widehat{\mathcal{A}}_e(\boldsymbol{U}_h, \nabla\boldsymbol{U}_h)\Big) : [\![\boldsymbol{\varphi}]\!]_e,\end{aligned} \tag{6}$$

where $\{\boldsymbol{V}\}_e = \frac{1}{2}(\boldsymbol{V}^+ + \boldsymbol{V}^-)$ denotes the average and $[\![\boldsymbol{V}]\!]_e = (\boldsymbol{n}^+ \otimes \boldsymbol{V}^+ + \boldsymbol{n}^- \otimes \boldsymbol{V}^-)$ the jump of the discontinuous function $\boldsymbol{V} \in V_h$ over element boundaries. For matrices $\sigma, \tau \in \mathbb{R}^{m\times n}$ we use standard notation $\sigma : \tau = \sum_{j=1}^{m}\sum_{l=1}^{n} \sigma_{jl}\tau_{jl}$. Additionally, for vectors $\boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^n$, we define $\boldsymbol{v}\otimes\boldsymbol{w} \in \mathbb{R}^{m\times n}$ according to $(\boldsymbol{v}\otimes\boldsymbol{w})_{jl} = \boldsymbol{v}_j\boldsymbol{w}_l$ for $1 \le j \le m$, $1 \le l \le n$.

The convective numerical flux $\widehat{\mathcal{F}}_e$ can be any appropriate numerical flux known for standard finite volume methods. Thus $\widehat{\mathcal{F}}_e$ could be simply the local Lax-Friedrichs flux function or a more problem tailored flux (i.e. approximate Riemann solvers) known from finite volume methods (Kröner [1997]). A wide range of diffusion fluxes $\widehat{\mathcal{A}}_e$ can be found in the literature, for a summary see (Arnold et al. [2002]). Many of these fluxes are available within this module as well as newer variations, e.g., the CDG2 flux which was shown to be highly efficient for the Navier-Stokes equations (cf. Brdar et al. [2012a]).

2.2 Temporal discretization

To solve the time dependent problem (1) we use a method of lines approach in which the DG method described above is first used to discretize the spatial operator and then a solver for ordinary differential equations is used for the time discretization. After spatial discretization, the discrete solution $\boldsymbol{U}_h(t) \in V_h$ has the form $\boldsymbol{U}_h(t,x) = \sum_i \boldsymbol{U}_i(t)\boldsymbol{\varphi}_i(x)$. We get a system of ODEs for the coefficients of $\boldsymbol{U}(t)$ which reads

$$\boldsymbol{U}'(t) \quad = \quad f(\boldsymbol{U}(t)) \text{ in } (0,T] \tag{7}$$

with $f(\boldsymbol{U}(t)) = M^{-1}\mathcal{L}_h(\boldsymbol{U}_h(t))$, M being the mass matrix which is in our case block diagonal or even the identity, depending on the choice of basis functions. $\boldsymbol{U}(0)$ is given by the projection of $\boldsymbol{U}_0$ onto V_h.

A range of different ODE solvers are available most based around *Strong Stability Preserving* Runge-Kutta methods (SSP-RK). Explicit methods up to forth order are available as well as implicit and semi-implicit Runge-Kutta solvers based on a Jacobian-free Newton-Krylov method (see Knoll and Keyes [2004]). All these methods are available through the Dune-Fem framework. The results and implementation techniques presented in this paper can be applied to explicit, implicit, or semi-implicit methods and mostly a **matrix-free** implementation of the discrete operator $\mathcal{L}_h$ is used. In addition, assembled operators are available.

3 Implementation

It is well known that numerical tools for solving partial differential equations strongly depend on the type of the equations. In order to unify all of the numerical tools and types of partial differential equations in a modular toolbox, a common interface is needed. The aim of this section is to give a short overview on the implementation of central classes of DUNE-FEM-DG. All the classes described below are used by the examples in section 4.

3.1 Simulator: Starting a Simulation

Simulations are usually started inside a `main.cc` file where the structure for each simulation looks very similar.

The most important steps therein are

- including a header with a user defined algorithm creator, which is explained in subsection 3.3.1, and
- starting the simulation with the `Simulator`.

Of course there are some other steps that should be done to run a DUNE-FEM-DG simulation properly. Thus, a `main.cc` should at least contain the following lines[2]

C++ code

```
1  // configure macros
2  #include <config.h>
3  // include a simulator which is able to call the algorithm creator
4  #include <dune/fem-dg/misc/simulator.hh>
5  // include a user defined algorithm creator
6  #include "algorithmcreator.hh"
7
8  int main(int argc, char** argv)
9  {
10   // Initialize MPI (always do this even if you are not using MPI)
11   Dune::Fem::MPIManager::initialize(argc, argv);
12   try
13   {
14     // append parameters from command line and remove from argv list
15     Dune::Fem::Parameter::append(argc,argv);
16     // if parameter file is given append that
17     if( argc >= 2)
18       Dune::Fem::Parameter::append(argv[1]);
19     else
20       // read default parameter file
21       Dune::Fem::Parameter::append("parameter");
22
23     // select the grid type
24     typedef Dune::GridSelector::GridType GridType;
25     // define an algorithm via a algorithm creator
26     Dune::Fem::MyAlgorithmCreator<GridType> algorithmCreator;
27
28     // run simulation
29     Dune::Fem::Simulator::run(algorithmCreator);
30   }
31   catch(...)
32   {
33     std::cerr << "Generic exception!" << std::endl;
34     return 1;
35   }
36   return 0;
37 }
```

[2] By `My...` we denote user defined classes/structs which should be replaced by an appropriate class/struct or just a user defined name.

The `Simulator` runs the simulation for a globally defined polynomial order `POLORDER`. Instead of just defining one polynomial order, it is also possible to provide a range between `MIN_POLORD` and `MAX_POLORD`. In the case `MIN_POLORD < MAX_POLORD` the polynomial order can be selected dynamically. For hp-adaptive simulations, this globally defined polynomial order is the lowest possible order for the p-refinement.

C++ code

```
#define POLORDER = 1
#define MIN_POLORD = POLORDER
#define MAX_POLORD = POLORDER

template <int polOrd, class Problem>
inline void simulate(const Problem& problem)
{
  // create pointer to grid
  typedef typename Problem::GridType GridType;
  std::unique_ptr<GridType> gridptr(problem.initializeGrid().release());
  // create algorithm
  typedef typename ProblemTraits::template Algorithm<polOrd> AlgorithmType;
  std::unique_ptr<AlgorithmType> algorithm(new AlgorithmType(*gridptr));

  // call compute method
  compute(*algorithm);
}

template <int polOrd>
struct SimulatePolOrd
{
  template <class Problem>
  static void apply(const Problem& problem, const int pOrder, const bool compAnyway)
  {
    if(compAnyway || polOrd == pOrder)
      simulate<polOrd> (problem);
  }
};

struct Simulator
{
  template <class Problem>
  static void run(const Problem& problem)
  {
    // dynamic parameter, usually read through a parameter file
    int polOrder = 1;

    // run through all available polynomial order and check with dynamic polOrder
    typedef Dune::ForLoop<SimulatePolOrd, MIN_POLORD, MAX_POLORD> ForLoopType;
    ForLoopType::apply( problem, polOrder, bool(MIN_POLORD == MAX_POLORD) );
  }
};
```

The `Simulator` only provides a static member function `run()` which calls a template function `simulate()` for the desired polynomial degree. This is done via a static for loop and a helper class `SimulatePolOrd`. Inside the `simulate()` function, the algorithm is created and the free `compute()` function is called which is explained in the following section.

3.2 Algorithm and SubAlgorithm

An *algorithm* (in the DUNE-FEM-DG sense) is a class derived from the interface class called `AlgorithmInterface`. It contains a `solve(int eocLoop)` method which is called by the free function `compute(Algorithm& algorithm)`. The aim of the `compute(Algorithm& algorithm)` function is to run the algorithm for different refinements of the grid which is needed for EOC

(experimental order of convergence) calculations. The avoidance of EOC calculations is accomplished via a single execution of the EOC loop (i.e. `algorithm.eocParams().steps() == 1`) and can thus be handled as a special case of a general EOC calculation loop.

C++ code

```
template <class Algorithm>
void compute(Algorithm& algorithm)
{
  // get the grid
  typedef typename Algorithm::GridType GridType;
  GridType& grid = algorithm.grid();

  // prepare data output
  auto dataTup = algorithm.dataTuple();
  typedef typename Algorithm::DataWriterCallerType::
    template DataOutput<GridType,decltype(dataTup)>::Type DataOutputType;
  DataOutputType dataOutput(grid, dataTup);

  // eoc loop
  for(int eocloop = 0; eocloop < algorithm.eocParams().steps(); ++eocloop)
  {
    // call algorithm
    algorithm.solve(eocloop);

    // write solution to disc
    dataOutput.writeData(eocloop);

    // refine grid for next eoc step
    if(algorithm.eocParams().steps() > 1)
      GlobalRefine::apply(grid,Dune::DGFGridInfo<GridType>::refineStepsForHalf());
  }
}
```

Obviously, the requirements for an algorithm are small, which allows for solving a wide range of PDEs.

Until now, nothing is said about the kind of solution of an algorithm. The solution could be nearly arbitrary; DUNE-FEM-DG provides two[3] different kinds of algorithms:

- `EvolutionAlgorithm` for solving instationary PDEs in a time loop (method of lines),
- `SteadyStateAlgorithm` for solving stationary PDEs.

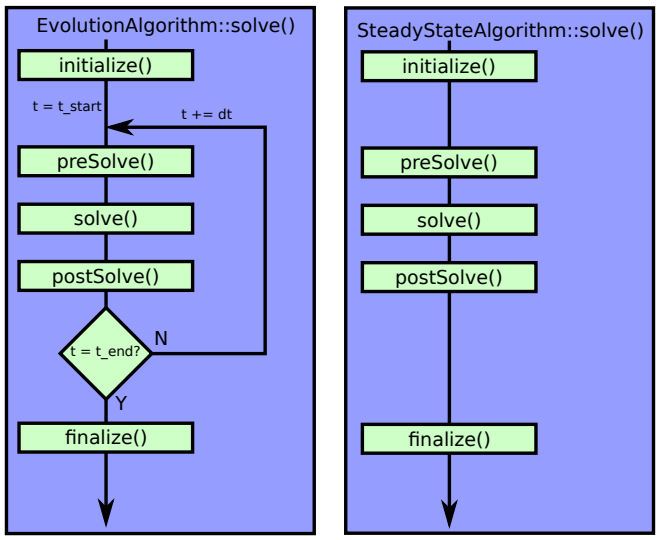
Although both algorithm classes are very different, it is possible to find a more detailed, common structure, which is visualized in Fig. 1. Each algorithm calls the same five methods:

1. `initialize(loop)`, run all the initializing steps which has to be executed once.
2. `preSolve(loop)`, run all the steps preparing the `solve()` step (for instationary algorithms: for each time loop).
3. `solve(loop)`, run a solver.
4. `postSolve(loop)`, run all the steps finalizing the `solve()` step (for instationary algorithms: for each time loop).
5. `finalize(loop)`, run all the finalizing steps which has to be executed once.

The idea is – since the methods are the same – that these five methods are implemented in an external class. We will call this external class a *sub-algorithm* which is given by the interface `SubAlgorihmInterface`. Again, there are several classes deriving from this interface class:

[3] More complex algorithms needed for simulation of multi physics will be presented in an upcoming paper.

Figure 1: The two central algorithm classes describing a instationary and a stationary PDE.



- `SubEvolutionAlgorithm` describing ingredients for an instationary PDE,
- `SubSteadyStateAlgorithm` describing a stationary PDE.

Furthermore, there are more specialized sub-algorithms for different kinds of PDEs:

- `SubEllipticAlgorithm`,
- `SubStokesAlgorithm`,
- `SubAdvectionAlgorithm` and
- `SubAdvectionDiffusionAlgorithm`.

Notice that with this scheme, it is also possible to plug a stationary sub-algorithm into a instationary algorithm.

In this paper we will focus on algorithms that only take *one* sub-algorithm. In an upcoming paper we will explain how algorithms taking several sub-algorithms could be implemented and used to solve multi physics problems.

In a nutshell: A sub-algorithm provides all the ingredients (i.e. the callback functions) needed by an algorithm. The algorithm decides how to use these ingredients (i.e. where and how to call the callback functions).

At first glance, the callback approach with the algorithm scheme presented in Fig. 1 may give the feeling that a simple problem is getting unnecessarily complicated. Our design decision always has to be seen in the background of multi physics problems where problems (and their implementations) can be very heterogeneous and thus an interface as small as possible is needed. Also keep in mind, that it is still possible to write specialized algorithms through the use of virtual inheritance[4].

Usually, callbacks suffer several disadvantages which can be overcome in a certain way:

[4]Furthermore, it is also possible to write an user defined algorithm which is not depending on sub-algorithms (and thus on callbacks) and incorporate all information from the sub-algorithms directly. It is clear that this would change the `SubAlgorithmCreator` described in section 3.3.1. Especially, for complex multi physic problems a generic algorithm cannot just be a simple composition of sub-algorithms: A coupling has to be implemented by the user.

- Isolation of data: Sub-algorithms only share less data which is mainly provided by the algorithm during the call of the callback function (time provider etc.) or through a global container class holding global data. There are two options for the case that sub-algorithms share much data: The simplest solution is to incorporate both sub-algorithms into one sub-algorithm. The second solution is to derive a user defined algorithm and provide more information, e.g. during the callback call.

- Black box design: Callbacks may be more complicated for the user because they may appear as a black box, where callback functions are defined somewhere else in the code. Thus, a clear documentation and a predefined location for the definition of callbacks are required. Most of the callbacks stick to a very similar patterns which reduces the complexity.

- Extendibility: The presented approach above is only a possible example. All important methods are virtual and can be overloaded.

- Increased work overhead: Although the two central algorithms may appear a little bit over simplified, they give a certain structure to the code.

3.2.1 Caller In order to incorporate further functionality such as adaptivity, solver monitoring, data I/O or limiter in each algorithm cycle DUNE-FEM-DG adds a `caller` concept. A common interface is provided by the `CallerInterface` class:

C++ code

```
1  struct CallerInterface
2  {
3    template<class... A> void initializeStart(A&&...) {}
4    template<class... A> void initializeEnd(A&&...) {}
5
6    template<class... A> void preSolveStart(A&&...) {}
7    template<class... A> void preSolveEnd(A&&...) {}
8
9    template<class... A> void solveStart(A&&...) {}
10   template<class... A> void solveEnd(A&&...) {}
11
12   template<class... A> void postSolveStart(A&&...) {}
13   template<class... A> void postSolveEnd(A&&...) {}
14
15   template<class... A> void finalizeStart(A&&...) {}
16   template<class... A> void finalizeEnd(A&&...) {}
17 };
```

Figure 2 shows how `Caller` work.

In DUNE-FEM-DG a predefined set of `callers` is implemented.

- `AdaptCaller`, it takes care of the adaptation procedure.

- `CheckPointCaller`, which takes care of backup/restore.

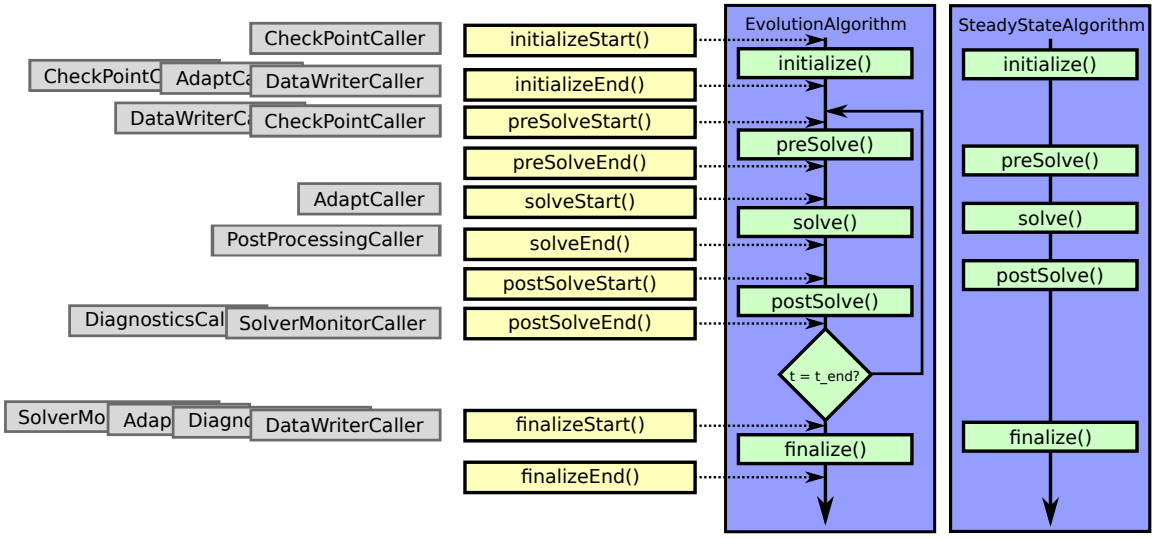
- `DataWriterCaller`, a caller which takes care of data output into different formats.

- `DiagnosticsCaller`, it collects diverse timing information, e.g. for load balancing and adaptation time.

- `PostProcessingCaller`, a caller which can be used to apply a limiter to the new solution.

- `SolverMonitorCaller`, this caller gathers information from the solvers.

Figure 2: Callers can be added at predefined locations inside the algorithm.



One basic and important assumption is the independency and self-containedness of each caller: Each caller should be optional, i.e. should be replaceable by a caller doing nothing without breaking the (general) validity of the whole algorithm.

At the moment, the selection of which callers to use in each `...Start()` and `...End()` call is defined by the algorithm class itself and cannot be changed without reimplementation of the algorithm. This is illustrated in Fig. 2 for the `EvolutionAlgorithm`. The `EvolutionAlgorithm` class uses a predefined set of callers which are defined in a traits class `EvolutionAlgorithmTraits`. Nevertheless, there is the possibility to exchange the caller implementation by using the more general `EvolutionAlgorithmBase` class.

Thus, instead of defining the `EvolutionAlgorithm`

C++ code

```
1 EvolutionAlgorithm<pOrd,UncoupledSubAlgorithms,MySubAlgorithmCreator>
```

we use

C++ code

```
1 EvolutionAlgorithmBase<MyEvolTraits<pOrd,MySubAlgorithmCreator>,UncoupledSubAlgorithms>
```

Here, `pOrd` is the polynomial degree of the algorithm, `UncoupledSubAlgorithms` is a place holder indicating that we are using simple, uncoupled sub-algorithm and the `MySubAlgorithmCreator` class will be described in subsection 3.3.1.

The `MyEvolTraits` class is a user defined traits class, which could look like the following:

C++ code

```
1 template<int pOrder,class Traits>
2 struct MyEvolTraits
3 {
4   // define grid type
5   typedef typename Traits::GridType                    GridType;
6
7   // define time provider
8   typedef GridTimeProvider<GridType>                   TimeProviderType;
9
```



```
10    // typedef of a sub-algorithm (hold in a tuple)
11    typedef std::tuple<typename std::add_pointer<Traits::template Algorithm<pOrder> > >
12                                                    SubAlgorithmTupleType;
13
14    // define callers: Either use a user defined implementation here or the
15    // predefined ones (empty template arguments '...Caller<>' means 'no caller')
16    typedef AdaptCaller<SubAlgorithmTupleType>          AdaptCallerType;
17    typedef CheckPointCaller<SubAlgorithmTupleType>     CheckPointCallerType;
18    typedef SolverMonitorCaller<SubAlgorithmTupleType>  SolverMonitorCallerType;
19    typedef DataWriterCaller<SubAlgorithmTupleType>     DataWriterCallerType;
20    typedef DiagnosticsCaller<SubAlgorithmTupleType>    DiagnosticsCallerType;
21    typedef PostProcessingCaller<SubAlgorithmTupleType> PostProcessingCallerType;
22
23    // extract tuple type of discrete functions that should be written to disk
24    typedef typename DataWriterCallerType::IOTupleType  IOTupleType;
25  };
```

Changing one of the caller definitions affects the behaviour of the algorithm.

Everything said about benefits and disadvantages of callbacks in the last section is also valid for the `Caller` classes. The main difference is that dynamic polymorphism is not used here (which is not caused by efficiency reasons):

1. The `Caller` classes should be as general as possible, i.e. even the arguments of the member functions are not prescribed by the interface class. Thus, we make use of the C++11 feature of variadic template member functions which cannot be virtual without specialization.
2. It is enough to write only one `Caller` class doing nothing for the 'deactivation' of a caller.

The usage of the `AdaptCaller`, `DataWriterCaller` and `CheckPointCaller` will be described more detailed later on.

3.3 AlgorithmCreator

As shown in the last subsection, the particular behaviour of an algorithm is defined via a sub-algorithm which is plugged into the algorithm class.

This combination of algorithm and sub-algorithm is set up inside the `AlgorithmCreator` class. An additional task of the `AlgorithmCreator` class is to define all the types of a sub-algorithm.

The rough structure of an algorithm creator could be the following.

C++ code

```
1   template<class GridImp>
2   struct MyAlgorithmCreator
3   {
4     // define a sub problem
5     struct MySubAlgorithmCreator{ /*==== SEE SUBSECTION 'SubAlgorithmCreator' ====*/ };
6
7     // define an algorithm: SteadyStateAlgorithm or EvolutionAlgorithm
8     template<int polOrd>
9     using Algorithm = MyAlgorithm<pOrd,UncoupledSubAlgorithms,MySubAlgorithmCreator>;
10
11    // type of the grid
12    typedef GridImp GridType;
13
14    // give each algorithm a unique name
15    static inline std::string moduleName()
16    {
17      return "myAlgorithm";
18    }
19
20    // create a grid and return a pointer to the grid
```



```
21    static inline GridPtr<GridType> initializeGrid()
22    {
23      return Fem::DefaultGridInitializer<GridType>::initialize();
24    }
25  }
```

The struct `MySubAlgorithmCreator` is a struct that contains several type definitions which in particular contains the definition of a sub-algorithm by defining a type alias `Algorithm`. The concrete structure of this struct is shown in the next subsection 3.3.1.

Furthermore, each algorithm creator contains a method to return a unique name and a method to create a grid (which returns the pointer to the newly created grid).

The classes `SteadyStateAlgorithm` and `EvolutionAlgorithm` (described in section 3.2) should fulfil all requirements to implement simple algorithms.

3.3.1 SubAlgorithmCreator The definition of a sub-algorithm is more comprehensive since DUNE-FEM-DG allows the exchange of a lot of different parts of the sub-algorithm.

C++ code

```
1   struct MySubAlgorithmCreator
2   {
3     /*===== SEE SUBSECTION 'AlgorithmConfigurator' =====*/
4     typedef MyAlgorithmConfiguratorType                          AC;
5
6     // define a grid type
7     typedef typename AC::GridType                                GridType;
8     // define a host grid part type (needed for moving grids)
9     typedef typename AC::GridParts                               HostGridPartType;
10    // define a grid part type
11    typedef HostGridPartType                                     GridPartType;
12
13    /*===== SEE SUBSECTION 'Problem' =====*/
14    typedef MyProblemInterfaceType                               ProblemInterfaceType;
15
16    // extract analytical function space from problem interface
17    typedef typename ProblemInterfaceType::FunctionSpaceType FunctionSpaceType;
18
19    // analytical problem descriptions
20    struct AnalyticalTraits
21    {
22      // define a problem type
23      typedef ProblemInterfaceType                               ProblemType;
24      // define a initial data type (usually contained in the problem type)
25      typedef ProblemInterfaceType                               InitialDataType;
26      /*===== SEE SUBSECTION 'Model' =====*/
27      typedef MyModelType                                        ModelType;
28
29      // define your error calculation here
30      template<class Solution,class Model,class ExactFunction,class TimeProvider>
31      static void addEOCErrors(TimeProvider& tp, Solution& u,
32                               Model& model, ExactFunction& f)
33      {}
34    };
35
36    // name the sub-algorithm in a unique name
37    static inline std::string moduleName()
38    {
39      return "mySubAlgorithm";
40    }
41
42    // return a problem which is derived from ProblemInterfaceType
43    static ProblemInterfaceType* problem()
44    {
45      /*===== SEE SUBSECTION 'Problem' =====*/
```



```
46      return MyProblemType();
47    }
48
49    template<int pOrd>
50    struct DiscreteTraits
51    {
52      // define type of a discrete function space
53      typedef typename AC::template DiscreteFunctionSpaces<GridPartType,pOrd,
54                                                           FunctionSpaceType>
55                                                             DFSpaceType;
56    public:
57      // type of discrete function
58      typedef typename AC::template DiscreteFunctions<DFSpaceType>
59                                                             DiscreteFunctionType;
60
61      // tuple of discrete functions for data I/O
62      typedef std::tuple<DiscreteFunctionType*>              IOTupleType;
63
64      // struct containing typedefs for the operators
65      struct Operator{ /*===== SEE SUBSECTION 'Operator' =====*/ };
66
67      // struct containing typedefs for a solver for the operator
68      struct Solver{ /*===== SEE SUBSECTION 'Solver' =====*/ };
69
70      // these classes are used by callers
71      // empty template arguments means: 'do nothing'
72      typedef AdaptIndicator<>              AdaptIndicatorType;
73      typedef SubSolverMonitor<>            SolverMonitorType;
74      typedef SubDiagnostics<>              DiagnosticsType;
75      typedef ExactSolutionOutput<>         AdditionalOutputType;
76    };
77
78    /*===== SEE SUBSECTION 'Algorithm and SubAlgorithm' =====*/
79    template< int polOrd >
80    using Algorithm = MySubAlgorithm<GridType,SubMyAlgorithmCreator,pOrd>;
81
82  };
```

Inside the `SubAlgorithmCreator` we have used several nested structs. The usage of nested structs has got two different aims:

1. grouping of type definition. This is used for `AnalyticalTraits` and `DiscreteTraits`, where we want to distinct between analytical and discrete description.
2. Allow a variable number of type definitions inside a struct for more flexibility which is done within the structs `Operator` and `Solver`.

For a `SubEvolutionAlgorithm` and a `SubSteadyStateAlgorithm` at least

C++ code

```
1  struct Operator
2  {
3    typedef MyOperator type;
4  };
```

and

C++ code

```
1  struct Solver
2  {
3    typedef MySolver type;
4  };
```


are required. For more complex operators, solvers and sub-algorithms it is possible to provide more type definitions inside these structs. Further details on the operator and solver structure will be given in the sections 3.3.5 and 3.3.6.

3.3.2 AlgorithmConfigurator A particular strength of DUNE-FEM-DG is the possibility to test different numerical schemes with different parameters. We have chosen different strategies to change these parameters to allow as much flexibility as possible:

- Static parameter selection: This could be either done with preprocessor defines in the `CMakeFile.txt` or the exchange of classes and type definitions in the `AlgorithmCreator`. While the first approach is the more global one, the second is more local, because it would also allow the usage of different classes in different sub-algorithms. Nevertheless, both alternatives have in common that they would need a recompilation of the source after changing one of the parameters.
- Dynamic parameter selection: This is usually done inside a parameter file which is called `parameter.in` and points to another parameter file called `parameter_cmake`. The first file only contains some `CMake` related variables. Changing one of these variables therein requires (for out-of-source builds) a reconfiguration of the module because the `parameter.in` file is copied to the build directory and has to be updated again. In order to avoid this, the `parameter_cmake` file is used for all non `CMake` related parameters. The usage of parameter files will be explained in subsection 3.5.1.

The task of an *algorithm configurator* is to hide as much template magic as possible from the user. The usage of an algorithm configurator in the `AlgorithmCreator` is not mandatory because it is still possible to use plain type definitions to set up the algorithm. It is even possible to have several algorithm configurators at the same time.

In the current implementation, the `AlgorithmConfigurator` takes nine template arguments:

- `GridType`, type of the grid all sub-algorithms have in common.
- `Galerkin::Enum`, an enum which defines the type of the Galerkin scheme,
  - `cg`: Continuous Galerkin and
  - `dg`: Discontinuous Galerkin.
- `Adaptivity::Enum`, an enum describing the adaptivity of the scheme,
  - `no`: no adaptation,
  - `yes`: adaptation possible.
- `DiscreteFunctionSpace::Enum`, an enum describing the discrete function space,
  - `lagrange`: discrete function space with Lagrange basis functions,
  - `legendre`: discrete function space with Legendre basis functions,
  - `hierarchic_legendre`: discrete function space with hierarchic Legendre basis functions,
  - `orthonormal`: discrete function space with orthornormal monomial basis functions.
- `Solver::Enum`, an enum describing the solver backend used for solving linear systems,
  - `fem`: use DUNE-FEM solver,
  - `femoem`: use matrix based version of DUNE-FEM solvers with `BLAS`,
  - `istl`: use DUNE-ISTL solver (Blatt and Bastian [2007]),

  - `umfpack`: use `UMFPACK` solver (Davis [2004]),
  - `petsc`: use `PETSc` solver (Balay et al. [2015, 1997]),
  - `eigen`: use `Eigen` solver (Guennebaud et al. [2010]).

- `AdvectionLimiter::Enum`, an enum describing the post processing (i.e. limiting) of the advection term,
  - `unlimited`: no limiting,
  - `limited`: limit the advection term.

- `Matrix::Enum`, an enum describing whether the system is assembled or not,[5]
  - `matrixfree`: use a matrix free operator,
  - `assembled`: use a assembled matrix describing the operator.

- `AdvectionFlux::Enum`, an enum describing numerical fluxes of the advective term,
  - `none`: no advection flux (no advection term),
  - `upwind`: Upwind flux,
  - `llf`: local Lax-Friedrichs flux,
  - `general`: dynamic numerical flux selection (standard) via parameter file,
  - `euler_llf`: local Lax-Friedrichs flux for Euler ("wellbalanced scheme"),
  - `euler_hll`: HLL flux specialized for Euler,
  - `euler_hllc`: HLLC flux specialized for Euler,
  - `euler_general`: dynamic numerical flux selection (for Euler) via parameter file.

- `DiffusionFlux::Enum`, an enum describing numerical fluxes of the diffusion term,
  - `none`: no diffusion (advection only) flux,
  - `cdg2`: CDG 2 (Compact Discontinuous Galerkin 2) flux,
  - `cdg`: CDG (Compact Discontinuous Galerkin) flux,
  - `br2`: BR2 (Bassi-Rebay 2) flux,
  - `ip`: IP (Interior Penalty) flux,
  - `nipg`: NIPG (Non-symmetric Interior Penalty) flux,
  - `bo`: BO (Baumann-Oden) flux,
  - `primal`: all of the above numerical fluxes selected via parameter file,
  - `br1`: BR1 (Bassi-Rebay 1) flux (local formulation only),
  - `ldg`: LDG (Local Discontinuous Galerkin) flux (local formulation only),
  - `local`: both, BR1 and LDG, via parameter file selection.

The functionality of the `AlgorithmConfigurator` may be expanded in future releases.

For the enum values `general` (or `euler_general`) and `primal` (or `local`) the advective and diffusive flux, respectively, has to be chosen inside the parameter file. For the diffusive flux, there are also some additional possibilities to adjust penalty terms and liftings of the flux.

[5] Usually, only one of the operators (assembled/matrix free) is implemented. This option can be seen as place holder for later projects.

Code

```
## choose a diffusion flux: CDG2, CDG, BR2, IP, NIPG, BO
dgdiffusionflux.method: CDG2
## scaling with theory parameters
dgdiffusionflux.theoryparameters: 1
## penalty factor
dgdiffusionflux.penalty: 0.
## scaling of the liftfactor
dgdiffusionflux.liftfactor: 1.0
## type of lifting: id_id, id_A and A_A
dgdiffusionflux.lifting: id_id
```

The advective flux can be selected with the following parameter.

Code

```
## choose an advective flux: NONE, UPWIND, LLF
## for Euler equations choose: EULER-LLF, EULER-HLL, EULER-HLLC, EULER-LLF2
dgadvectionflux: UPWIND
```

3.3.3 Model In DUNE-FEM-DG the analytical terms of problem (1) are implemented within a *model*. This model has to be implemented for each numerical example mentioned in 4 (user-defined, see 3).

We provide a common interface class `DefaultModel` which is able to capture all data which is needed by the examples presented in section 4.[6]

Each information needed to evaluate an analytical term at a given quadrature point is provided by a `LocalEvaluationContext`. The methods needed to form a `DefaultModel` are outlined in the following.

- For efficiency reasons a method indicating whether a flux term, e.g. either $\mathcal{F}(\boldsymbol{U})$ or $\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})$ is present. Otherwise the surface integration part can be skipped entirely.

C++ code

```
// indicates whether an advective or diffusive term is present
bool hasFlux() const;
```

- For the advective flux $\mathcal{F}(\boldsymbol{U})$ two methods need to be implemented.

C++ code

```
// evaluate advective flux F(U)
void advection(const LocalEvaluationContext& context,
               const RangeType& u,
               FluxRangeType& flux) const;

// evaluate advective boundary flux F(U) and return wave speed
double boundaryFlux(const LocalEvaluationContext& context,
                    const RangeType& u,
                    FluxRangeType& flux) const;
```

- For the term $\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})$ the diffusive flux implementation requires the follwing methods.

C++ code

```
// evaluate diffusive flux
void diffusion(const LocalEvaluationContext& context,
               const RangeType& u,
               const JacobianRangeType& jac,
               FluxRangeType& flux) const;
```

[6] For more complex equations it is also possible to expand this default model.


```
6
7  // evaluate diffusive boundary flux and return wave speed
8  double diffusionBoundaryFlux(const LocalEvaluationContext& context,
9                               const RangeType& u,
10                              const JacobianRangeType& jac,
11                              FluxRangeType& flux) const;
```

- The source term $\mathcal{S}(\boldsymbol{U})$ is split into a stiff and a non-stiff part. The following methods are required.

C++ code

```
1  // true if a (non-) stiff source exists
2  bool hasStiffSource() {}
3  bool hasNonStiffSource() {}
4
5  // evaluate stiff source term and return wave speed
6  double stiffSource(const LocalEvaluationContext& context,
7                     const RangeType& u,
8                     const JacobianRangeType& jac,
9                     RangeType& s) const;
10
11 // evaluate non stiff source and return wave speed
12 double nonStiffSource(const LocalEvaluationContext& context,
13                       const RangeType& u,
14                       RangeType& s) const;
```

- Dirichlet boundary conditions are treated using the following methods.
  Note that if `hasBoundaryValue()` returns false, meaning Neumann type boundary conditions, the corresponding boundary flux function is used.

C++ code

```
1  // return true if boundary values are provided for this quadrature point
2  bool hasBoundaryValue(const Intersection& intersection,
3                        const double time,
4                        const FaceDomainType& x) const;
5
6  // if boundary values are provided compute them
7  void boundaryValue(const Intersection& intersection,
8                     const double time,
9                     const FaceDomainType& x,
10                    const RangeType& u,
11                    RangeType& value) const;
```

- For time dependent problems the time needs to be known to compute the analytical terms. The following three methods are added to the model to deal with the time stages and time step estimations.

C++ code

```
1  // set current stage time
2  void setTime(const double time) {}
3
4  // evaluate the maximal wave speed of the advective term
5  void maxSpeed(const LocalEvaluationContext& context,
6                const DomainType& normal,
7                const RangeType& u,
8                double& advectionSpeed,
9                double& totalSpeed) const;
10
11
12 // provide a time step estimation for the the diffusive term
13 double diffusionTimeStep(const Intersection& intersection,
14                          const double elementVolume,
```



```
                          const double circumEsitimate,
                          const double time,
                          const FaceDomainType& x,
                          const RangeType& u) const;
```

For some numerical flux implementations like LDG or IP additional methods are required.

C++ code

```
// computes velocity in the given evaluation context
DomainType velocity(const LocalEvaluationContext& context) const;

// returns the maximum eigen value of A(U)
void eigenValues(const LocalEvaluationContext& context,
                 const RangeType& u,
                 RangeType& maxValue) const;

// evaluate the Jacobian of U (only used by local formulation)
void jacobian(const LocalEvaluationContext& context,
              const RangeType& u,
              JacobianRangeType& a) const;

// return penalty factor for diffusive fluxes
double penaltyFactor(double time,
                     const DomainType& x,
                     const EntityType& entity,
                     const RangeType& u) const;
```

For some mathematical models one might want a retardation term, for example, a factor multiplied to the mass term. In that case the following two methods need to be overloaded. The default implementation assumes a trivial mass term.

C++ code

```
// return true if mass term is non-trivial
bool hasMass() const;

// return mass term
inline void mass(const LocalEvaluationContext& context,
                 const RangeType& u,
                 MassRangeType& m ) const;
```

3.3.4 Problem In the last subsection 3.3.3 we have explained how analytical data of a PDE can be described by a model. Although a model would be enough to describe PDEs and data, Dune-Fem-DG introduces another layer: The usage of an additional *problem* class allows to create different test cases in an easy way. This problem class encapsulates data from the model, i.e. the model is collecting information from the problem class.

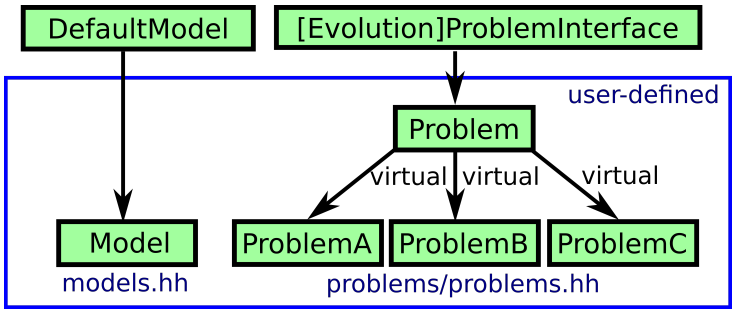
The type of a problem interface (or derived class) is known by the model class (defined inside the `SubAlgorithmCreator`). This is where the static `problem()` method of the `SubAlgorithmCreator` comes into play: Inside the sub-algorithm `SubAlgorithmInterface` a problem is created via the `problem()` method from the `SubAlgorithmCreator`.

C++ code

```
  Problem* problem = SubAlgorithmCreator::problem();
  Model model(*problem);
```

Basically, this allows to implement a number of methods in the *static polymorph model* using *dynamic polymorph problem* classes derived from a common pure virtual interface class. Fig. 3 visualizes the idea. Note that the *problem* is a template parameter for the *model* to also allow *static polymorphism* here, if necessary.

Figure 3: The `problem()` method in the `SubAlgorithmCreator` chooses one of the problems which are derived from a problem interface.



In DUNE-FEM-DG there are two basic problem interface classes

- `EvolutionProblemInterface`, a problem interface for instationary PDEs,
- `ProblemInterface`, a problem interface for stationary PDEs.

Since a model class is user-defined, it depends on the model implementation which data from the problem class is used. Nevertheless, there are commonalities between all PDEs presented in section 4 which can be described by a common problem class.

In the following, a short overview of the most important methods of the class `ProblemInterface`, which describes the most important factors of a (stationary) reactive advection-diffusion equation, are given. Non matching methods (e.g. a diffusion term for Euler equations) can – of course – be implemented in the problem class, but will be neglected by the model class.

C++ code

```
1  // getter for the velocity
2  virtual void velocity(const DomainType& x, DomainType& v) const;
3  // advection factor
4  virtual double constantAdvection() const;
5  // the diffusion matrix
6  virtual void K(const DomainType& x, DiffusionMatrixType& m) const;
7  // returns true if diffusion coefficient is constant
8  virtual bool constantK() const;
9  // mass factor gamma
10 virtual double gamma() const;
11 // the right hand side
12 virtual void f(const DomainType& x, RangeType& ret) const;
```

Boundary data is set by the following methods:

C++ code

```
1  // return whether boundary is Dirichlet (true) or Neumann (false)
2  virtual bool dirichletBoundary(const int bndId, const DomainType& x) const;
3  // the Neumann boundary data function
4  virtual void psi(const DomainType& x, JacobianRangeType& dn) const;
5  // the Dirichlet boundary data function
6  virtual void g(const DomainType& x, RangeType& ret) const;
```

For EOC calculation the exact solution has to be known.

C++ code

```
1  // evaluate the exact solution
2  void evaluate(const DomainType& x, RangeType& ret) const;
3  // the gradient of the exact solution
4  virtual void gradient(const DomainType& x, JacobianRangeType& grad) const;
```


Finally, some methods are implemented to give additional information, e.g.

C++ code

```
1 // latex output for eoc output
2 virtual std::string description() const;
```

Notice that the argument list of the methods is much shorter and simpler for the problems than for the models. For the `EvolutionProblemInterface` similar methods are defined.

3.3.5 Operator *Operators* in Dune-Fem-DG describe the discrete operator $\mathcal{L}_h$ from equation (4). In Dune-Fem-DG operators are derived from `Fem::SpaceOperatorInterface`. This is a Dune-Fem-DG with a pure virtual `operator()` implementing an operator between two discrete spaces. This module provides four operators:

- `DGLimitedAdvectionOperator`,
- `DGDiffusionOperator`,
- `DGAdvectionDiffusionOperator`,
- `DGLimitedAdvectionDiffusionOperator`.

The *operators* are currently based on the pass concept presented in (Dedner et al. [2010b]). We do not want to go into more details of the pass concept since the three operators provide the functionality needed for the tests described in this paper. Further examples where passes have been used for handling moving domains can be found in Klöfkorn and Nolte [2014] .

The operators mentioned so far are combined with matrix free solvers. To solve linear versions of (3) Dune-Fem-DG also provides classes for assembling matrices.

There are currently two assemblers in Dune-Fem-DG: `DGPrimalMatrixAssembly` for the assembly of operators of the form (2) and a class `StokesAssembler`, which assembles the Stokes problem. Assembled operators follow the interfaces for linear operators from Dune-Fem and the available solver interfaces can be used.

In an upcoming paper we will present more operators which are not based on the pass concept.

3.3.6 Solver *Solvers* in the Dune-Fem concept are inverse operators.

For the solution of the linear algebra system, different solvers are available. The solvers using the Dune-Istl module and the `PETSc` framework can be modified inside the parameter file.

Code

```
1  ########## DUNE-ISTL ###############
2  ## dune-Istl solvers
3  istl.preconditioning.method: amg-ilu-0
4  ## number of precondition iterations
5  istl.preconditioning.iterations: 1
6  ## number of relaxation steps
7  istl.preconditioning.relaxation: 1
8
9  ########## PETSc ####################
10 ## PETSc solver: cg, bicg, bicgstab, gmres
11 petsc.kspsolver.method: cg
12 ## PETSc preconditioning: asm, sor, jacobi, hypre, ml, ilu-n, lu, icc, mumps, superlu
13 petsc.preconditioning.method: none
14 ## number of precondition iterations
15 petsc.preconditioning.iterations: 5
```


For instationary problems the `RungeKuttaSolver` class is used which bundles the DUNE-FEM classes `ExplicitRungeKuttaSolver` (explicit RK schemes), `ImplicitRungeKuttaSolver` (implicit RK schemes) and `SemiImplicitRungeKuttaSolver` (implicit/explicit RK schemes). The scheme (and other related parameters) can be selected in the parameter file (via EX, IM, IMEX, respectively).

Code

```
## type of ode solver: EX, IM, IMEX
fem.ode.odesolver: EX
## set order of ode solver
fem.ode.order: 2
## set verbose mode of ode output: none, cfl, full
fem.ode.verbose: none
## initial estimation of cfl number
fem.ode.cflStart: 1.
## factor for cfl number on increase (decrease is 0.5)
fem.ode.cflincrease: 1.25
## number of minimal iterations that the linear solver should do
## if the number of iterations done is smaller, then the cfl number is increased
fem.ode.miniterations: 95
## number of maximal iterations that the linear solver should do
## if the number of iterations is larger, then the cfl number is decreased
fem.ode.maxiterations: 105
## maximum cfl number
fem.ode.cflMax: 5
```

Inside an implicit Runge-Kutta step the solution of a nonlinear equation might be necessary. This is done using the Newton scheme implemented in DUNE-FEM (`NewtonInverseOperator`). The following parameters in the parameter file may influence the Newton solver:

Code

```
## print debug output for Newton solver, 0 = no
fem.solver.newton.verbose: 0
## set maximum number of linear iterations in each Newton step
fem.solver.newton.maxlineariterations: 1000
## set solver tolerance of Newton solver
fem.solver.newton.tolerance: 1e-10
```

3.3.7 Linkage between Sub-Algorithms, Solvers and Operators

Sub-algorithms, solvers and operators are closely linked because the solution of many PDEs requires specialized solvers and thus also a specialized set of (discrete) operators. In the following, we want to give some examples on this topic.

The class `SubAdvectionDiffusionAlgorithm` describes a governing equation (1) and uses a Runge-Kutta solver for the solution of the equation. In order to apply an explicit, implicit or semi-implicit Runge-Kutta solver, the splitting of the operator $\mathcal{L}_h = \mathcal{L}_{\text{impl},h} + \mathcal{L}_{\text{expl},h}$ into an implicit operator $\mathcal{L}_{\text{impl},h}$ and an explicit operator $\mathcal{L}_{\text{expl},h}$ has to be defined. Furthermore, we want to be able to exchange the linear solver backend. For the `Solver` and `Operator` struct in the algorithm creator the following type definitions are mandatory.

C++ code

```
struct Operator
{
  // defines the full operator
  typedef typename AC::template Operators<MyOpTraits,OperatorSplit::Enum::full>
                                                        type;
  // defines the explicit operator
  typedef typename AC::template Operators<MyOpTraits,OperatorSplit::Enum::expl>
                                                        ExplicitType;
  // defines the implicit operator
  typedef typename AC::template Operators<MyOpTraits,OperatorSplit::Enum::impl>
```



```
                                                    ImplicitType;
};
struct Solver
{
  // defines the linear solver
  typedef typename AC::template LinearSolvers<DFSpaceType>  LinearSolverType;
  // defines the ode solver
  typedef DuneODE::OdeSolverInterface<DiscreteFunctionType> type;
};
```

The `SubAdvectionAlgorithm` models a governing equation (1) with $\mathcal{A} = 0$. Everything said for the `SubAdvectionDiffusionAlgorithm` is also true for the class `SubAdvectionAlgorithm`, except that we suppose $\mathcal{L}_h = \mathcal{L}_{\text{expl},h}$. Thus, it is sufficient to provide only the full operator $\mathcal{L}_h$.

The class `SubPoissonAlgorithm` models a stationary Poisson equation. The operator $\mathcal{L}_h$ is linear and can be written as $\mathcal{L}_h(U) := A(u) - S$, where A is a matrix and S a right hand side. Here, we distinguish between the linear operator A and the assembler which is able to set up the matrix and right hand side entries. For the `Solver` and `Operator` struct in the algorithm creator the following type definitions are mandatory

C++ code

```
struct Operator
{
  // defines the assembler
  typedef typename AC::template Operators<MyOpTraits> AssemblerType;
  // defines a linear operator
  typedef typename AssemblerType::LinearOperatorType  type;
};
struct Solver
{
  // defines a linear solver
  typedef typename AC::template LinearSolvers< DFSpaceType > type;
};
```

The class `SubStokesAlgorithm` is similar to the `SubPoissonAlgorithm`, but uses an `UzawaSolver` and internally holds the type of an elliptic algorithm, i.e. the `SubEllipticAlgorithm` class.

Table 1 gives an overview on all DUNE-FEM-DG sub-algorithms and their required type definitions.

Sub-Algorithm	Temporal Solver	Spatial Solver	Operator
`SubEvolutionAlgorithm`	`::type`	–	–
`SubAdvectionAlgorithm`	`::type`†	`::LinearSolverType`	`::type`

`SubAdvectionDiffusionAlgorithm`	`::type`†	`::LinearSolverType`	`::type`
			`::ImplicitType`
			`::ExplicitType`

`SubSteadyStateAlgorithm`	–	`::type`	`::type`
`SubEllipticAlgorithm`	–	`::type`	`::type` `::AssemblerType`
`SubStokesAlgorithm`	–	`::type`	`::type` `::AssemblerType`

Table 1: This table lists some of the relations between sub-algorithm, solver and operators for the current implementation: Types starting with `::` are defined inside the structs `Operator` and `Solver` of the algorithm creator, respectively. The type definitions marked with † are only the base classes: The method `doCreateSolver()` returns a shared pointer to `RungeKuttaSolver` and thus uses dynamic polymorphism.

3.4 Adaptivity

Adaptation is realized using the caller class `AdaptationCaller`. The `AdaptationCaller` collects the `AdaptIndicator` from the sub-algorithm and manages the adaptation process.

The main purpose of an adapt indicator is to estimate the local error and to mark the elements which should be refined or coarsened.

C++ code

```
1 void estimateMark(const bool initialAdapt = false) {}
```

The `AdaptIndicator` supports different refinement strategies, e.g. a shock indicator (see Dedner and Klöfkorn [2011]) or a gradient based indicator.

The adaptation method (i.e. the marking strategy) can be chosen in the parameter file. A lot of other parameters can help to adjust the adaptation method:

Code

```
1  ## marking strategy
2  ## shockind = shock indicator,
3  ## apost    = a posteriori based indicator,
4  ## grad     = gradient based indicator
5  fem.adaptation.markingStrategy: apost
6  ## choose an adaptation method: none | generic | callback
7  fem.adaptation.method: none
8  ## specify refinement tolerance
9  fem.adaptation.refineTolerance: 0.5
10 ## percent of refinement tol used for coarsening
11 fem.adaptation.coarsenPercent: 0.05
12 ## coarsest level that should be present
13 fem.adaptation.coarsestLevel: 0
14 ## finest level that should be present
15 fem.adaptation.finestLevel: 8
16 ## adaptation call after 'adaptcount' many time step,
17 ## 0 disables adaptation
18 fem.adaptation.adaptcount: 1
```

3.5 Data Input/Output and Checkpointing

In DUNE-FEM-DG data input and output is used for two purposes: reading simulation parameters and writing simulation data to disk for post-processing and checkpointing.

3.5.1 Input Simulation parameters are read by using the `Parameter` class from DUNE-FEM. This allows an easy but very flexible parameter input from a parameter file or the command line. The `Parameter` class uses a singleton concept to ease the use of parameter reading, i.e. no specific object has to be created and dragged around. Defining a parameter that should be set by a parameter file looks like this:

C++ code

```
1   const double defaultValue = 1.0;
2   // read value from parameter file or command line and use default value if not found
3   double value = Dune::Fem::Parameter::getValue<double>("value", defaultValue);
```

The parameter value can either be a fixed value, read from another parameter file, the output of a bash command, derived by an other parameter value or the combination of all of them.

Code

```
## value (default = 1)
value: 0.5

## set full path of all parameter files
fem.prefix.in: full/path/to/all/paramfiles
## read additional parameters from another parameter file
paramfile: relative/path/to/a/paramfile

## NOTE: Set to 1 for parameter substitution (see next lines)
fem.resolvevariables: 1

## use return value of running <command> as parameter value
commandValue: $[ <command> ]

## use parameter value of a different parameter
derivedValue: $(value)

## combined parameter, call <command> with '$(value) + 1' as argument
combinedValue: $[ <command> $(value) + 1 ]
```

If the parameter should be overloaded on the command line we write `value:0.5`, i.e. no white-space between the keyword and the value. In DUNE-FEM-DG first the command line is parsed for parameters and then the provided parameter file is parsed. Parameter values are set with the value obtained by the first found matching keyword.

3.5.2 Output Simulation data are written to disk using the `DataWriter` class from DUNE-FEM. A `DataWriterCaller` caller class is used to integrate the data writer functionality into the algorithms. DUNE-FEM's data writer classes are controlled with the following parameters.

Code

```
## gives file prefix for each data file written to disk
fem.prefix: "path/for/data-ouput"
## this adds a file prefix for each written data file
fem.io.datafileprefix: "file-prefix"
## defines the type of output format, possible formats:
## binary, vtk-cell, vtk-vertex, gnuplot, sub-vtk-cell
fem.io.outputformat: vtk-cell
## this gives the number of 'virtual' refinements for 'sub-vtk-cell'
fem.io.subsamplinglevel: 0

## write data every 'saveStep' time period, <=0 deactivates
fem.io.savestep: 0.00001
## write data every saveCount time steps, <=0 deactivates
fem.io.savecount: 42
```

The `vtk` output can be visualized using a `vtk`-reader like `ParaView` (Ahrens et al. [2005]). An extra feature of DUNE-FEM-DG's data output functionality is the possibility to write additional data to disk. This can be either a projection of the exact solution or a derived quantity. The data output caller class collects additional output from a sub-algorithm. `AdditionalOutput` is specified within the discrete traits section of the sub-algorithm creator, see section 3.3.1 for further details.

3.5.3 CheckPointing In the last subsection 3.5 we have explained how numerical solutions are written to disk. A further important aspect of data I/O is checkpointing, which is described in the following. DUNE-FEM-DG provides a checkpointing functionality from DUNE-FEM which is done in the class `CheckPointer`. This class provides a checkpointing functionality for writing and reading data to and from disk enabling a program to resume from a previously saved state. All objects registered to this class are saved when a checkpoint is written. On restart, the data is restored consistently.

In Dune-Fem-DG we use the `CheckPointCaller` caller class to incorporate the checkpointing feature into the algorithms. A small helper class `GridCheckPointCaller`, which is not a real caller, has to be used to construct the grid in the `initializeGrid()` method of the algorithm creator.

Furthermore, it is important to mention that many grid manager do not support backup and restore routines of the Dune-Grid interface, yet. That is why Dune-ALUGrid should be the first choice for checkpointing.

Checkpointing is controlled inside the parameter file. The following parameters are read:

Code

```
## if the following variable is specified, the simulation
## is started from the last checkpoint, otherwise it is started from the beginning
fem.io.checkpointrestartfile: "path/to/checkpoint/"
## write checkpoint every 'checkpointstep' time step
fem.io.checkpointstep: 42
## write 'checkpointmax' number of different checkpoints (ring buffer)
## existing checkpoints are overridden, if maximum number of checkpoints is reached
fem.io.checkpointmax: 1
```

3.6 Parallelization

The parallelization techniques in Dune (Bastian et al. [2008a,b]) and Dune-Fem (Dedner et al. [2010b]) are based on domain decomposition using MPI [2009] for data exchange between multiple processes. The domain decomposition is usually achieved by using graph partitioning tools like ParMETIS (Schloegel et al. [2001]) and depends on the selected grid implementation. In Dune, for example, `ALUGrid` (Alkämper et al. [2016]), or `SPGrid` (Klöfkorn and Nolte [2012]) can be used for parallel computation. More parallel grid implementations are available in the Dune-Grid module.

During the evaluation of the discrete operator $\mathcal{L}_h$ in (4) numerical fluxes at cell boundaries have to be evaluated. i.e. when computing $\mathcal{I}_h$ in (6). This means that for one element the information about the solution $\boldsymbol{U}_h$ on directly neighboring cells is needed. If the neighboring cell is a ghost cell, communication has to be used to obtain data of the solution $\boldsymbol{U}_h$ on this cell. In fact, for the DG method it would be sufficient to only exchange values of the discrete function $\boldsymbol{U}_h$ at the process boundaries since for the evaluation of $\mathcal{I}_h$ neighboring information is only needed at the cell interfaces and not on the neighboring cell itself. Theoretically, this allows to completely avoid ghost cells. However, the corresponding communication interfaces for exchanging data on an intersection e are still missing in Dune. Therefore, we have to rely on the ghost cell approach for the evaluation of numerical fluxes at process boundaries.

Using the Dune grid interface communication, a natural way to exchange data for the evaluation of the discrete spatial operator would be an interior-ghost communication just before the evaluation of the discrete spatial operator. This guarantees that all necessary data for the evaluation of the numerical fluxes are present. We call this *synchronous communication* in the sense that every process has to wait until all communications have been finished before starting with the computation of $\mathcal{L}_h$.

Dune-Fem-DG also supports *asynchronous communication* and *shared memory* parallelization. For both features the shared memory parallelization has to be enabled, even if only one thread is used for computation. This is done by setting the `cmake` variables `USE_OPENMP=ON` or `USE_PTHREADS=ON`.

The basic idea of *asynchronous communication* is to hide network latency behind the evaluation of the element integrals since these integrals can be computed without information from other partitions. To achieve this, we use the splitting of the discrete operator into element and surface integrals, i.e. $\mathcal{L}_h(\boldsymbol{U}_h) = \mathcal{K}_h(\boldsymbol{U}_h) + \mathcal{I}_h(\boldsymbol{U}_h)$, from equation (4). Since for the computation of $\mathcal{K}_h$ (given in equation (5)) on each cell K no neighboring information is needed and thus no data exchange between different processes is necessary. The improved computation of $\mathcal{L}_h$ is done in

the following steps: (i) send interior data required by other processes, (ii) compute $\mathcal{K}_h(\boldsymbol{U}_h)$ given in (5), (iii) wait until all data is received, (iv) compute $\mathcal{I}_h(\boldsymbol{U}_h)$ given in (6), and (v) finally compute $\mathcal{L}_h(\boldsymbol{U}_h)$ as given in (4) which is only a vector operation. Further details are given in (Klöfkorn [2012]) including a strong scaling study for the presented DG solvers.

For *shared memory* or hybrid parallelization we simply split the set of elements for computation of $\mathcal{L}_h$ into $|\mathcal{G}|/N_{threads}$ chunks (± 1) and avoid race conditions by two sided computing of numerical fluxes at thread domain boundaries (see Klöfkorn [2012]).

The number of threads to be used and communication type is set in the parameter files:

Code

```
## compute on 4 threads
fem.parallel.numberofthreads: 4
## write speedup diagnostics file (0 = no, 1 = yes)
fem.parallel.diagnostics: 1
## if true non-blocking communication is enabled
femdg.nonblockingcomm: true
```

3.7 Code generation

A huge amount of time in numerical schemes is spent during the evaluation of the DG basis functions in quadrature points. Although values of the scalar DG basis functions are cached in Dune-Fem for each quadrature point, multiplication with values e.g. degrees-of-freedom is needed in order to evaluate analytical terms mentioned in section 3.3.3. Those evaluations can be seen as dense matrix-matrix multiplication. For a-priori known sizes of theses matrices modern CPU structures such as AVX or SSE can be used efficiently to increase code performance. For known combinations of basis function sets and quadratures at compile time automatic code generation is employed to help the *compiler* optimizing the matrix-matrix multiplication and thus the evaluation of discrete functions. In Dune-Fem-DG this is done in two steps:

First, the program is analyzed to obtain which combinations of basis functions and quadrature rules are used. This is achieved by creating a special target, indicated by the compiler definition `BASEFUNCTIONSET_CODEGEN_GENERATE`. This compiler switch enables the code analyzer in the `Simulator`. The code analyzer is usually started with the following parameters:

Code

```
fem.eoc.steps:1
femdg.stepper.maximaltimesteps:1
fem.io.outputformat:none
```

This will run the ordinary (non optimized) simulation for only one time step. During this run a call of `axpy()`, `evaluateAll()` and `jacobianAll()` methods from the basis function will lead to the creation of a new plain C++ header file inside a new folder called `autogeneratedcode`. These header files contain classes which partially specializes the corresponding template structs `EvaluateJacobians`, `EvaluateRanges`, `AxpyRanges` and `AxpyJacobians`. In the end, the file `autogeneratedcode.hh` is generated which includes all header files from the `autogeneratedcode` folder.

Second, optimized code for these combinations is generated. The optimized target is built using the compiler definition `USE_BASEFUNCTIONSET_CODEGEN`. This replaces the evaluation of the default basis functions.

A `CMake` macro takes care of this steps. Applied to a given target `my_target` the following additional `CMake`-targets are created:

- `my_target_codegenerator` is a code analyzer that tracks each combination of basis function and quadrature.

- `my_target_generate` executes the analyzer and generates the source code.
- `my_target_optimized` is the final target which includes automatic generated code.

In order to enable code generation in new executables, the `CMake` macro

Code

```
add_code_generate_targets(my_target)
```

has to be used in the `CMakeList.txt` which will create the custom target `my_target_generate` and the two executables `my_target_codegenerator` and `my_target_optimized`.

In summary, the following three command line arguments are needed to generate and run the optimized code.

Bash code

```
my_target_generate
my_target_optimized
./mytarget_optimized
```

3.7.1 Floating point operations per second Floating point operations per second (FLOPS) during a program run can be counted with tools like `likwid` (see Treibig et al. [2010]) or the library `PAPI` (see Terpstra et al. [2009] and Weaver and Dongarra [2010]) which is available through DUNE-FEM. `likwid` can be used without program modification and for `PAPI` we provide the parameter

Code

```
femdg.flopcounter: true
```

to enable FLOPs counting with `PAPI`. Both tools, however, depend on hardware counters for floating point operations. Newer CPUs like the Intel Haswell series do not provide these counters for floating point operations to the extend needed.

Because of that, DUNE-FEM provides an overloaded version of the standard `double` that is called `Dune::Fem::Double`. To enable FLOPs counting in DUNE-FEM-DG the preprocessor variable `COUNT_FLOPS` has to be set and the `GRIDTYPE` should be set to `SPGRID_COUNT_FLOPS` from the DUNE-SPGRID package available at https://gitlab.dune-project.org/extensions/dune-spgrid. This grid implementation allows to change the coordinate type. In Section 4.1 we provide some performance evaluation. Using this optimized automatically generated code, rates close to 30% of the performance of the Intel Linpack benchmark can be achieved (see Section 4.1). Note, that floating point measures computed using `Dune::Fem::Double` present a lower bound since standard functions like `std::pow` or `std::sin` are not accounted for.

4 Numerical examples

In the following we show different types of time dependent and steady state advection-diffusion problems, which were successfully solved with DUNE-FEM-DG. In appendix C we will show how the reader can reproduce the numerical results presented in the following. All convergence studies presented in the following are carried out in a unit cube domain $\Omega := [0,1]^d$ with a grid spacing of 4 cells in each direction and a refinement bisecting the grid width for each step. The DGF grid files are `unitcube2.dgf` and `unitcube3.dgf`.

Test case	v	$\mu(\boldsymbol{U})$
(*TC0*): Heat equation	user defined	$\mu(\boldsymbol{U}) = \varepsilon$
(*TC1*): C^∞ problem	$v = (1, ..., 1)^{\mathrm{T}}$	$\mu(\boldsymbol{U}) = \varepsilon$
(*TC2*): Quasi-Heat equation	$v = 0$	$\mu(\boldsymbol{U}) = \lvert\boldsymbol{U}\rvert\varepsilon$
(*TC3*): Pulse problem	$v(x) = (-4x_2, 4x_1, 0, ...)^{\mathrm{T}}$	$\mu(\boldsymbol{U}) = \varepsilon$

Table 2: This table lists the test settings for the advection-diffusion example 4.1. They can be chosen via run time parameter `problem` in the parameter file.

grid width		$\lVert\boldsymbol{U} - \boldsymbol{U}_h\rVert_{L^2}$	EOC		$\lVert\boldsymbol{U} - \boldsymbol{U}_h\rVert_{L^2}$	EOC
0.25	(*TC0*)	0.00526937	—	(*TC1*)	0.0158056	—
0.125		0.000932952	2.50		0.00154145	3.36
0.0625		0.000121377	2.94		0.000219992	2.81
0.03125		$1.54114e-05$	2.95		$2.99987e-05$	2.87
0.25	(*TC2*)	$1.97242e-05$	—	(*TC3*)	0.0307972	—
0.125		$3.57044e-06$	2.47		0.00954669	1.69
0.0625		$4.95925e-07$	2.85		0.00132529	2.85
0.03125		$6.39905e-08$	2.95		0.000153224	3.11

Table 3: This table lists the errors for the general advection diffusion reaction problem. The user defined data are chosen as $\varepsilon = 0.001$ and end time $T = 1$. For each test scenario (*TC0* - *TC3*) we observe an experimental order of convergence of about 3.

4.1 Advection-Diffusion equation

In this example we consider a general advection-diffusion problem as described in equation (1). The vector of conservative variables is $\boldsymbol{U} = (u)^{\mathrm{T}}$. $\mathcal{F}(\boldsymbol{U})$, $\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})$ and $\mathcal{S}(\boldsymbol{U})$ are given as follows:

$$\mathcal{F}(\boldsymbol{U}) = \begin{pmatrix} v_1 u \\ \vdots \\ v_d u \end{pmatrix}, \qquad \mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U}) = \mu(\boldsymbol{U}) \begin{pmatrix} \partial_1 u \\ \vdots \\ \partial_d u \end{pmatrix}, \qquad \mathcal{S}(\boldsymbol{U}) = f, \tag{8}$$

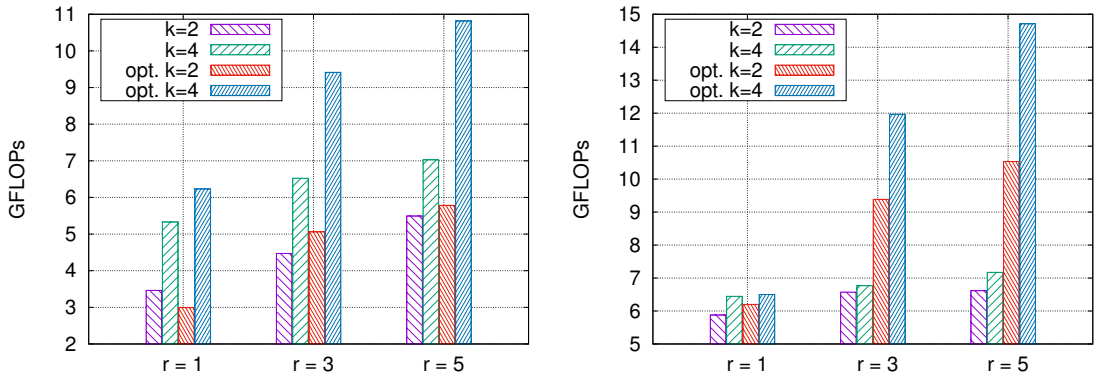
where $\mu(\boldsymbol{U})$, v and f are defined within each test case.

The following four test cases are provided to cover a wide range of possible applications. In Table 2 we present different choices of v and $\mu(\boldsymbol{U})$. ε is a user defined constant. User defined means that the corresponding parameter is chosen within the parameter file. In each test case the boundary conditions and the source function f are defined in correspondence to a given exact solution.

In Table 3 we present the results of our numerical experiments. The error between a given exact solution $\boldsymbol{U}$ and the numerical solution $\boldsymbol{U}_h$ is computed in the L^2-norm. In each test scenario the ansatz space was chosen to have order 2.

To demonstrate the performance of the implementation we count the number of floating point operations during the entire program run. We run 100 timesteps for test case 3 (TC3) using the `LegendreDiscontinuousGalerkinSpace` for polynomial orders 2 and 4 for different dimensions of the range space, e.g. 1, 3, and 5. For both spatial dimensions we run the experiments on 4 096 elements and 4 threads. The number of floating point operations is computed in a separate run using `Dune::Fem::Double` as field type which overloads all double operations and counts each operation in addition. The number of floating point is then divided by the total runtime used to compute the 100 timesteps including setup time.

To compare the performance of the DG implementation we compute a *peak* number of floating point operations with the Intel Linpack Benchmark at version 11.3.3.011. On average this benchmark yields ≈ 50 GFLOPs on the Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz. In Figure 4 we present different floating point measurements for TC3. We can see that for 2d and a low order and low dimensional range space the code generation (marked with opt. in the graphs) does not lead to an improved performance. However, for 3d and vector valued problems ($r = 3, 5$) we observe a significant performance improvement. The best performance, 14.71 GFLOPs, is observed in 3d for $k = 4$ and $r = 5$ using the auto generated code. This result corresponds to about 30% of the reported peak performance. Note that this setting is equivalent to the consideration of the compressible Navier-Stokes equations in 3d where similar performance results have been reported earlier in Klöfkorn [2012].



(a) 2d (b) 3d

Figure 4: 2d results (left) and 3d results (right) for test case 3 (TC3) using different polynomial orders and different dimensions of the range space.

4.2 Euler equation

In this example we consider the Euler equations of gas dynamics which have the form

$$\partial_t \boldsymbol{U} + \nabla \cdot \mathcal{F}(\boldsymbol{U}) \quad = \quad 0 \qquad \text{in } (0,T] \times (\Omega \subset \mathbb{R}^d) \tag{9}$$

with suitable initial and boundary conditions which corresponds to (1) if we choose $\mathcal{A}(\boldsymbol{U}, \nabla \boldsymbol{U}) = 0$ and $\mathcal{S}(\boldsymbol{U}) = 0$.

The vector of conservative variables is $\boldsymbol{U} = (\rho, \rho \boldsymbol{v}, \rho E)^{\mathrm{T}}$. ρ is the density, ρE is the total energy density, and $\boldsymbol{v} = (v_1, ..., v_d)^T$ is the velocity field. $\mathcal{F}(\boldsymbol{U}) = (\mathcal{F}_1(\boldsymbol{U}), ..., \mathcal{F}_d(\boldsymbol{U}))$ which for $j = 1, ..., d$ has the form:

$$\mathcal{F}_j(\boldsymbol{U}) = \begin{pmatrix} \rho v_j \\ \rho v_1 v_j + \delta_{1j} p \\ \vdots \\ \rho v_d v_j + \delta_{dj} p \\ (E + p) v_j \end{pmatrix} \tag{10}$$

The system is closed by the equation of state for an *ideal gas* where the pressure is given by

$$p(\boldsymbol{U}) \quad = \quad (\gamma - 1)\Big[E - \frac{\rho}{2}|\boldsymbol{v}|^2\Big],$$

where γ is the ratio of specific heat. For an ideal gas the *speed of sound* c_s and *Mach number* M are given by the formulae

$$c_s(\rho, p) = \sqrt{\gamma \frac{p}{\rho}}, \qquad M = \frac{v}{c_s}, \tag{11}$$

where v is the speed of the considered fluid, i.e. the gas.

For the atmospherics applications a slightly different system is implemented using the potential temperature as a conservative variable instead of the total energy. However, these examples described in Brdar et al. [2013], Schuster et al. [2014], Dedner and Klöfkorn [2016], Klöfkorn [2012], Brdar et al. [2011b], and Brdar et al. [2011a] are contained in a separate module.

For higher order Discontinuous Galerkin approximations we apply a shock detector combined with a slope limiter for stabilization of the scheme. This stabilization scheme is described in detail in Klöfkorn [2009], Dedner and Klöfkorn [2011] and acts in an element by element fashion. If a shock situation is detected the polynomial degree of the numerical solution is reduced to at most linear and a limiter function to reduce the slope, if necessary, is applied. Note, that in addition to the DG solution reconstructions can be computed improving the resolution of the scheme in shock regions (Klöfkorn [2009], Dedner and Klöfkorn [2011]). When choosing a constant ansatz space the scheme yields a second order finite volume scheme. The implementation is available in Dune-Fem-DG and further details on the limiting procedure can be found in Klöfkorn [2009], Dedner and Klöfkorn [2011].

For a finer tuning of the limiting process the following parameters can be used:

Code

```
## 0 = only dg solution | 1 = only reconstruction | 2 = both
femdg.limiter.admissiblefunctions: 1
## tolerance for shock indicator
## (for cells with values below the solution will limited)
femdg.limiter.tolerance: 1
## threshold for avoiding over-excessive limitation
femdg.limiter.limiteps: 1e-8
## add indicator to outputvariables
femdg.limiter.indicatoroutput: true
```

In Fig. 5 we present the results (taken from Klöfkorn [2009], Dedner and Klöfkorn [2011]) for the 3d Forward Facing Step example. A third order DG scheme with stabilization and local grid adaptivity is used. The scheme works very well in parallel environments. A detailed convergence study can be found in Klöfkorn [2009], Dedner and Klöfkorn [2011].

4.3 Compressible Navier-Stokes equation

In addition to the Euler problem, presented in the last section, we show the application of Dune-Fem-DG to the compressible Navier-Stokes equation. Let $\boldsymbol{U}$, $\mathcal{F}(\boldsymbol{U})$, $\mathcal{S}(\boldsymbol{U})$, and p be as in section 4.2. The diffusion term $\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})$ for the compressible Navier-Stokes equation is given as follows for $j = 1, ..., d$:

$$\Big(\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})\Big)_j = \begin{pmatrix} 0 \\ \tau_{j1} \\ \vdots \\ \tau_{jd} \\ \sum_{l=1}^{d} \tau_{jl} v_l + \lambda \partial_j T \end{pmatrix}. \tag{12}$$

with

$$\tau_{jl} = 2\eta D_{jl}(\boldsymbol{U}) - \frac{2}{3}\eta \delta_{jl} \nabla \cdot \boldsymbol{v} \qquad \text{and} \qquad D_{jl}(\boldsymbol{U}) = \frac{1}{2}(\partial_l v_j + \partial_j v_l) \tag{13}$$

with $D(\boldsymbol{U})$ the deformation tensor, the temperature T, the dynamic shear viscosity η, and the thermal conductivity λ.

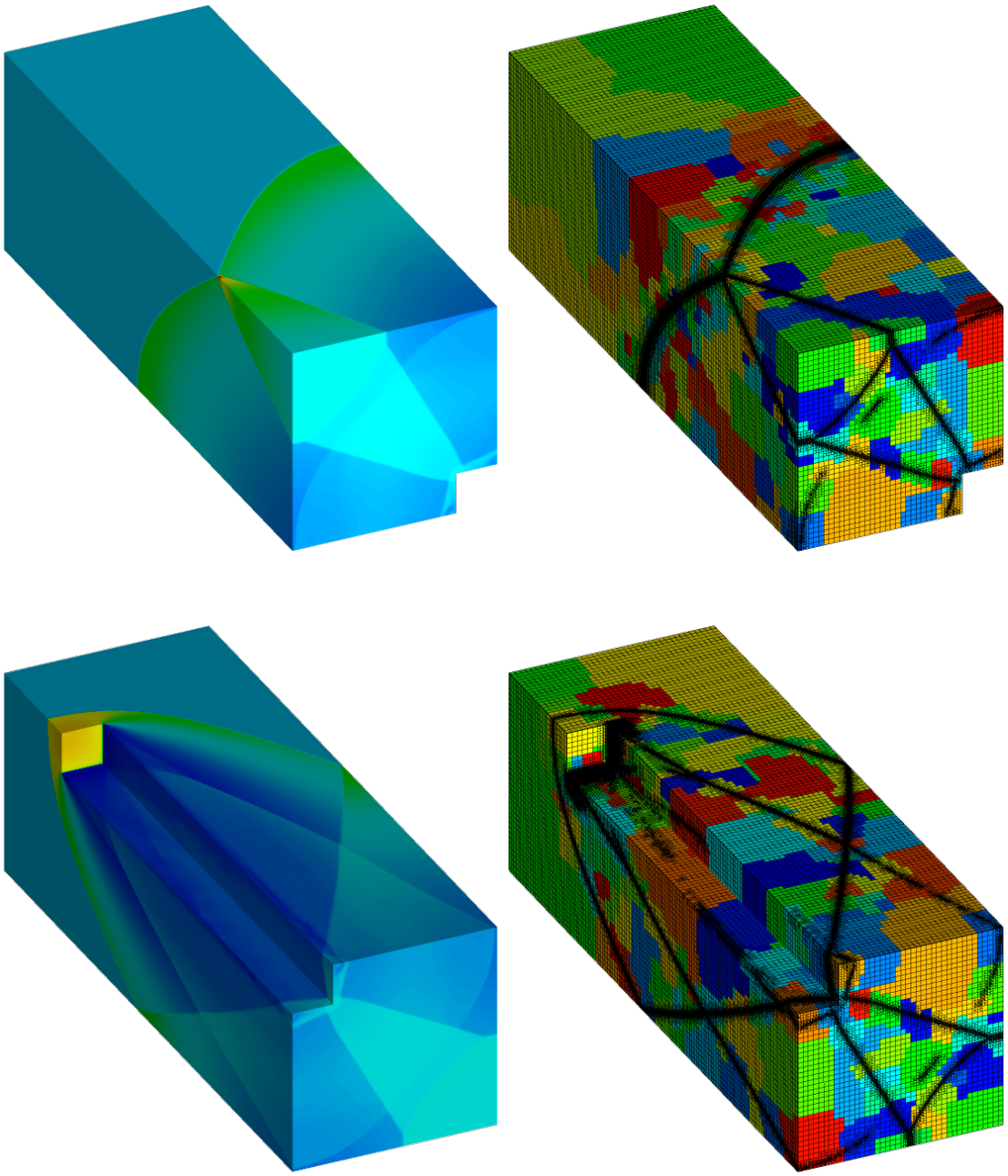


Figure 5: Density distribution obtained with the stabilized DG scheme, adapted grid and partitioning of the grid at time $t = 2$ for test the adaptive Forward Facing Step 3d (see Dedner and Klöfkorn [2011] for details). The calculation used `ALUCubeGrid` and 512 processors. Quadratic basis functions (k=2) have been used. The initial grid contains 185 856 hexahedrons and the final grid contains about 4.5 million hexahedrons.

As a test case we choose a setting with given exact solution. The solution $\boldsymbol{U}$ and the source term $\mathcal{S}(\boldsymbol{U})$ are taken from [Gassner, 2009, Appendix F]. Dirichlet boundary conditions are applied. Further applications in atmospheric flow (Brdar et al. [2013], Schuster et al. [2014]) and reactive flow (Klöfkorn and Nolte [2014]) have been considered.

	2d		3d	
grid width	$\|\|\boldsymbol{U} - \boldsymbol{U}_h\|\|_{L^2}$	EOC	$\|\|\boldsymbol{U} - \boldsymbol{U}_h\|\|_{L^2}$	EOC
0.25	0.00201777	—	0.00217464	—
0.125	0.000389584	2.37	0.00040635	2.42
0.0625	$6.32243e-05$	2.62	$6.8199e-05$	2.58
0.03125	$6.74915e-06$	3.23		

Table 4: This table lists the errors for the compressible Navier Stokes poblem. The end time is chosen to be $T = 0.01$. We observe an experimental order of convergence of about 2.5 for the 2d and 3d test.

grid width	$\|\|\boldsymbol{U} - \boldsymbol{U}_h\|\|_{L^2}$	EOC	$\|\|\boldsymbol{U} - \boldsymbol{U}_h\|\|_{\mathrm{DG}}$	EOC
0.25	0.000106058	—	0.00574261	—
0.125	$1.45255e-05$	2.87	0.00142307	2.01
0.0625	$1.8948e-06$	2.94	0.000354109	2.01
0.03125	$2.41794e-07$	2.97	$8.83146e-05$	2.00

Table 5: In this table the errors for test case 1, defined in Dedner and Klöfkorn [2008], for the Poisson problem are presented. The first column shows the maximal grid width. In the second and third column we list the L^2-error between $\boldsymbol{U}$ and $\boldsymbol{U}_h$ and its experimental order of convergence. The last two columns list the DG error and its EOC.

4.4 Poisson equation

In the previous sections time dependent problems were discussed. This sections will show the application of Dune-Fem-DG to a stationary problem: the Poisson equation. Let $\boldsymbol{U} = (u)^{\mathrm{T}}$ be the vector of variables. The Poisson equation is given as follows:

$$-\nabla \cdot \Big(\mathcal{A}(\boldsymbol{U}, \nabla \boldsymbol{U})\Big) \;=\; \mathcal{S}(\boldsymbol{U}) \qquad \text{in } \Omega \tag{14}$$

with suitable boundary conditions. $\mathcal{A}(\boldsymbol{U}, \nabla \boldsymbol{U})$ is given as follows:

$$\mathcal{A}(\boldsymbol{U}, \nabla \boldsymbol{U}) = K \begin{pmatrix} \partial_1 u \\ \vdots \\ \partial_d u \end{pmatrix}. \tag{15}$$

with $K \in \mathbb{R}^{d \times d}$ the diffusion matrix. The source term is $S(\boldsymbol{U}) = f$, with f a given function. Many test cases for (an-)isotropic diffusion problems can be found in the literature. In Table 5 we present results for test case 1 taken from Klöfkorn [2011]. We show the error reduction for the L^2- and DG-error. Further results can be found in Eymard et al. [2011] and Klöfkorn [2011].

In Fig. 6 we present the solution of the Fichera corner problem computed on a 3d L-shape grid. The adaptive algorithm and error estimator used to compute the solution as well as the DG-norm are described in Dedner et al. [2014].

4.5 Stokes equation

The vector of conservative variables is $\boldsymbol{U} = (\boldsymbol{v}, p)^{\mathrm{T}}$, where p is the pressure and $\boldsymbol{v} = (v_1, ..., v_d)^{\mathrm{T}}$ the velocity field. The Stokes equation is given as follows:

$$\nabla \cdot \Big(\mathcal{F}(\boldsymbol{U}) - \mathcal{A}(\boldsymbol{U}, \nabla \boldsymbol{U})\Big) \;=\; \mathcal{S}(\boldsymbol{U}) \qquad \text{in } \Omega \tag{16}$$

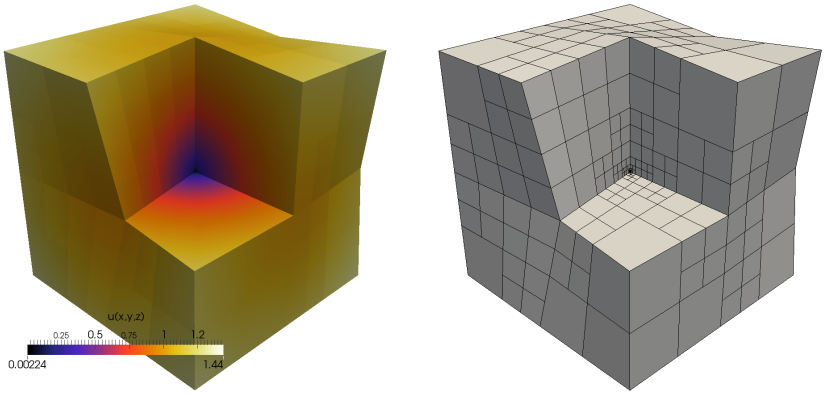


Figure 6: Left, the computed solution u_h at iteration 11 of the adaptive algorithm. Right, the corresponding refined hexahedral grid with non-affine geometry mapping.

grid width	$\|v - v_h\|_{L^2}$	EOC	$\|v - v_h\|_{\mathrm{DG}}$	EOC	$\|p - p_h\|_{L^2}$	EOC
0.25	0.0158905	—	0.82312	—	0.129294	—
0.125	0.00217847	2.87	0.208366	1.98	0.0265494	2.28
0.0625	0.000282386	2.94	0.0521036	1.99	0.00451457	2.56
0.03125	$3.61808e-05$	2.96	0.0130128	2.00	0.000829281	2.44

Table 6: This table lists the errors for the first Stokes test case. In the first column we list the maximal grid width, the second and third column show the L^2-error of the velocity and its Experimental Order of Convergence (EOC). The forth and fifth column lists the DG-error of the velocity and its EOC. In the last two columns we present the L^2-error and its EOC of the pressure.

with suitable boundary conditions. $\mathcal{F}(\boldsymbol{U}) = (\mathcal{F}_i(\boldsymbol{U}))$ and $\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U}) = ((\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U}))_i)$ for $i = 1, ..., d$ are given as follows:

$$\mathcal{F}_i(\boldsymbol{U}) = \begin{pmatrix} \delta_{1i}p \\ \vdots \\ \delta_{di}p \\ v_i \end{pmatrix}, \qquad \Big(\mathcal{A}(\boldsymbol{U}, \nabla\boldsymbol{U})\Big)_i = \mu \begin{pmatrix} \partial_i v_1 \\ \vdots \\ \partial_i v_d \end{pmatrix}, \tag{17}$$

with μ being the kinematic viscosity. The source term is $S(\boldsymbol{U}) = (f_1, ..., f_d, 0)^{\mathrm{T}}$, with given functions $f_1, ..., f_d$. The first test case is chosen to verify the approximation quality for $d = 2$. An exact solution is given, namely

$$\boldsymbol{U}(x, y) = \sin(2\pi(x + y))\,(1, -1)^{\mathrm{T}}, \qquad p(x, y) = \sin(2\pi(x - y)),$$

for which the boundary conditions, kinematic viscosity and right hand side function are computed. In Table 6 we present the errors between the exact solution $\boldsymbol{U}$ and the discrete solution $\boldsymbol{U}_h$, which is computed in a Discontinuous Galerkin Taylor-Hood space of order 2. The error is measured in the L^2-norm for the velocity and pressure component. Additionally, the DG-error for the velocity component is computed. The velocity errors behave as expected. For the pressure we see a super convergence effect.

As a second test case we have taken the driven cavity problem. On the domain $\Omega = [0, 1]^2$ we set

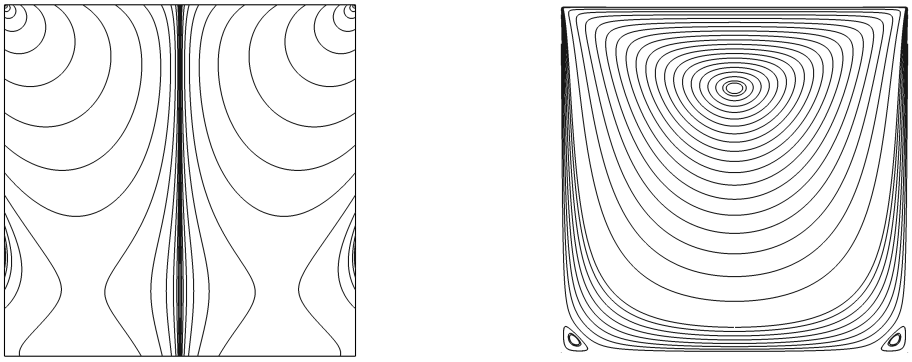


Figure 7: This figure show the numerical solution to the driven cavity test problem. Left: pressure contour lines. Right: stream traces. A huge vortex within the cavity and the two smaller recirculation areas in the lower bottom part can be identified. The simulation was carried out on a structured mesh with about 102400 elements. A Taylor-Hood $\mathcal{P}_2^2 - \mathcal{P}_1$ ansatz space is used.

$\mathcal{S}(U) = 0$ and Dirichlet boundary conditions:

$$v(x) = \begin{cases} (2,0) & \text{if } x \in [0,1] \times [1] \\ 0 & \text{else} \end{cases}$$

In Fig. 7 we present the numerical results for $\mu = 1.0$.

5 Summary

We have shown that the newly released Dune-Fem-DG module is able to solve a wide range of problems using Discontinuous Galerkin methods. A lot of scientific papers have used the Dune-Fem-DG module in the past and our goal is to enlarge the class of problems which can be solved.

One of the next steps for Dune-Fem-DG is the simulation of PDEs stemming from multi physics problems. Two examples of multi physics we are currently interested in are the fluid-structure interaction problem and the simulation of atherosclerotic plaque formation (a chronic inflammation of the blood vessel wall which may lead to a heart attack, see Girke et al. [2014]).

Fig. 8 and 9 show a draft on how the implementation could be realized within the Dune-Fem-DG framework. The boxes with the brightest blue describe the algorithm concept from section 3.2 realizing the main time loop. The dark blue colored boxes represent the sub-algorithm concept where each problem is solved in a monolithic fashion. One of the simplest couplings is the consecutive call of all sub-algorithms within the algorithm. Once the user has specified which solution of each sub-algorithm are associated, the class `EvolutionAlgorithm` automatically calls all sub-algorithms consecutively. This also includes the call of all callers mentioned in section 3.2.1.

Although it is possible to write more generic coupling classes (e.g. a simple fixed-point iteration) it may become cumbersome at some point. The usual way to build more complex couplings would be to derive from the `EvolutionAlgorithm` and to write a user-defined algorithm, especially by overloading the `solve()` method. In Fig. 8 and 9 this is indicated by the medium blue colored boxes.

Figure 8: Atherosclerotic Plaque. A detailed simulation of atherosclerotic plaque can be roughly divided into three main parts – blood flow **(A)**, inflammation **(B)** and deformation (of the vessel wall due to plaque) **(C)** – and contains the solution of different PDEs, defined on different domains and different time scales. The main factor for atherosclerosis is thought to be the penetration of Low Density Proteins (LDL) from the blood vessel into to vessel wall. This penetration is mainly influenced by the wall shear stress of the blood onto the vessel wall. Thus, a Navier-Stokes equation has to be solved in the blood vessel and a Darcy equation in the vessel wall. This is coupled to an advection-diffusion equation for the LDL via a membrane equation. During each heart beat the local blood flow changes which makes a very small time step size necessary **(A)**. On the other hand, the inflammation of the vessel wall is modeled using a reactive advection-diffusion equation for different species involved in this inflammation **(B)**. This inflammation leads to a volume increase of the vessel wall **(C)** but only large time steps are sufficient as atherosclerotic plaque growth over years and decades.

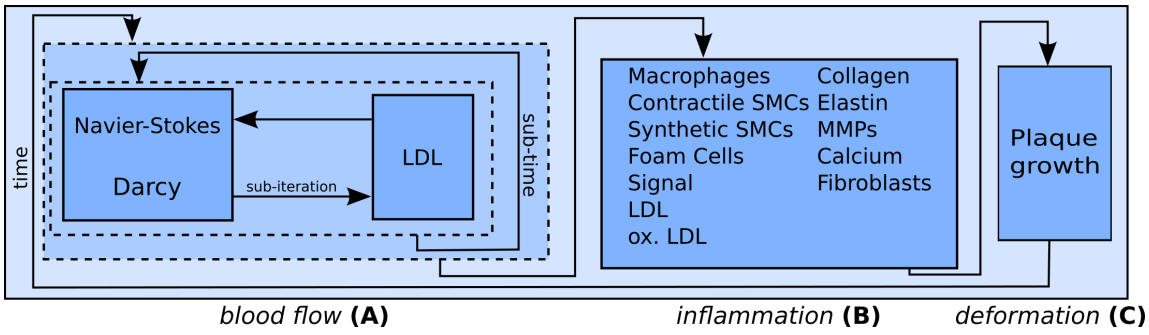
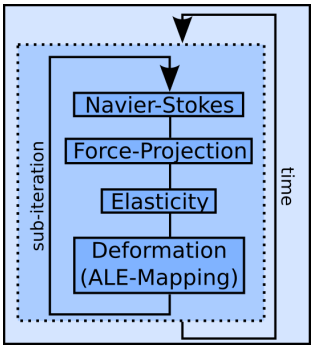


Figure 9: Fluid-Structure Interaction. Different approaches have been developed in the recent years to solve the fluid-structure interaction problem: While one approach is the monolithic one where the whole problem is solved at once, another idea is to use segregated solvers. For each time step a fixed point iteration is started: first solving the Navier-Stokes equation in the fluid domain, then projecting the force onto the boundary of the solid domain, then solving the elasticity equations on the solid domain and finally computing the resulting deformation. This is done until a threshold is reached.



The implementation of these two applications is ongoing work and may result in small interface changes for future releases. One should also mention that a major benefit of this modular implementation is the exchangeability of each sub module. This would allow to consecutively improve mathematical models. One assumption for the simulation of atherosclerotic plaque (see Fig. 8) is that only deformation of the vessel wall occurs due to plaque growth (on a long time scale). This assumption neglects the deformation of the vessel wall due to each heart beat.

Using the work of the fluid-structure interaction problem could help to easily improve the current simulation of atherosclerotic plaque.

Acknowledgements Stefan Girke was supported by the Deutsche Forschungsgemeinschaft, Collaborative Research Center SFB 656 "Cardiovascular Molecular Imaging", project B07, Münster, Germany.

Robert Klöfkorn acknowledges the Research Council of Norway and the industry partners – ConocoPhillips Skandinavia AS, BP Norge AS, Det Norske Oljeselskap AS, Eni Norge AS, Maersk Oil Norway AS, DONG Energy A/S, Denmark, Statoil Petroleum AS, ENGIE E&P NORGE AS, Lundin Norway AS, Halliburton AS, Schlumberger Norge AS, Wintershall Norge AS – of The National IOR Centre of Norway for financial support.

A License

The DUNE-FEM-DG module is available under the GNU General Public License version 2, or (at your option), any later version.

B Installation notes

The DUNE-FEM-DG module is available under https://gitlab.dune-project.org/dune-fem/dune-fem-dg. The version used in this paper can be downloaded with the command

Bash code

```
git clone --branch releases/2.4 \
https://gitlab.dune-project.org/dune-fem/dune-fem-dg.git
```

After downloading the DUNE-FEM-DG module run

Bash code

```
./dune-fem-dg/scripts/build-dune-fem-dg.sh
```

from the command line which will

- download all depending DUNE modules,
- run `dunecontrol` (build all DUNE modules),
- run the `CMake` test build system and produce all tests.

By default the `CMake` test build system will only print whether a test has passed or failed. To see the whole output on the command line, it is possible to use the verbose output of `ctest` directly.

Bash code

```
cd dune-fem-dg/scripts/dune-fem-dg/
ctest -V
```

Another approach is to directly build and modify the examples which can be found in the directory `dune-fem-dg/scripts/dune-fem-dg/dune/fem-dg/examples/`.

The `CMakeLists.txt` and parameter files are located in the corresponding `test` folder.

C Case Studies

In this section we show how DUNE-FEM-DG can be used to reproduce the results presented in section 4. We assume that DUNE-FEM-DG is build and configured as explained in the previous section. All paths used in the following description are relative to the `CMake` build directory `dune-fem-dg`. For each numerical example DUNE-FEM-DG writes numerical solutions into the `test/data` sub directory located in the examples directory. Additional information such as grid width, errors, EOC, number of iterations/timesteps or run-time are printed onto screen. Unless otherwise specified each example is build for grid dimension $d = 2$, polynomial order $k = 2$ and `ALUGrid< cube >` as the default grid manager. To change either the grid dimension, grid manager or the polynomial order the corresponding `CMake` definition

Code

```
1  set( GRIDTYPE YASPGRID )
2  set( GRIDDIM 2 )
3  set( POLORDER 2 )
```

has to be set to the desired value in the source directory file `CMakeLists.txt`. A reconfiguration and recompilation of the DUNE-FEM-DG module is necessary to apply the changes.

- Advection-Diffusion example:
  In section 4.1 four test cases have been presented. The simulation for a test case is done by calling:

  Bash code

  ```
  1  cd dune/fem-dg/examples/advdiff/test
  2  ./advdiff "problem:<problem>"
  ```

  with <problem> being one of the following parameters: `heat` *(TC0)*, `sin` *(TC1)*, `quasi` *(TC2)* or `pulse` *(TC3)*. Default is test case *(TC2)*.

- Compressible Euler example:
  In section 4.2 several test cases have been presented. The simulation for a test case is done by calling:

  Bash code

  ```
  1  cd dune/fem-dg/examples/euler/test
  2  ./euler "problem:<problem>"
  ```

  with <problem> being one of the following parameters: `sod`, `riemann`, `ffs`, `smooth1d`, or `schockbubble`. The default is test case `sod`.

- Compressible Navier-Stokes example:
  In section 4.3 one test case was presented. The simulation for a test case is done by calling:

  Bash code

  ```
  1  cd dune/fem-dg/examples/navierstokess/test
  2  ./navierstokes
  ```

- Poisson example:
  For the Poisson example the results presented in 4.4 can be reproduced calling in a terminal:

  Bash code

  ```
  1  cd dune/fem-dg/examples/poisson/test
  2  ./poisson
  ```

  Further problems are available for the Poisson problem which can be chosen via parameter `problem`, see poisson example parameter file for further test cases.

- Stokes example:
  The results for the two test cases presented in 4.5 for the Stokes example can be reproduced by calling

Bash code

```
cd dune/fem-dg/examples/stokes/test
./stokes "problem:<problem-number>"
```

  with <problem-number> 0 for test scenario 1, which is the default value, or <problem-number> 1 for the driven cavity problem. The contour lines for the pressure and the vorticity are generated using ParaView in a post-processing step, which has to be done manually.

References

MPI: A Message-Passing Interface Standard. Version 2.2. High Performance Computing Center Stuttgart (HLRS), 2009.

J. Ahrens, B. Geveci, C. Law, C. Hansen, and C. Johnson. 36-paraview: An end-user tool for large-data visualization, 2005.

M. Alkämper, A. Dedner, R. Klöfkorn, and M. Nolte. The DUNE-ALUGrid Module. *Archive of Numerical Software*, 4(1):1–28, 2016. URL http://dx.doi.org/10.11588/ans.2016.1.23252.

D. Arnold, F. Brezzi, B. Cockburn, and L. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, 2002.

S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015. URL http://www.mcs.anl.gov/petsc.

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007. URL http://dealii.org/.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2–3):121–138, 2008a. URL http://dx.doi.org/10.1007/s00607-008-0004-9.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008b. URL http://dx.doi.org/10.1007/s00607-008-0003-x.

P. Bastian, F. Heimann, and S. Marnach. Generic implementation of finite element methods in the Distributed and Unified Numerics Environment (DUNE). *Kybernetika*, 46:294–315, 2010.

M. Blatt and P. Bastian. The iterative solver template library. In B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing – State of the Art in Scientific Computing*, pages 666–675, Berlin/Heidelberg, 2007. Springer.

S. Brdar. *A higher order locally adaptive discontinuous Galerkin approach for atmospheric simulations*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2012. https://www.freidok.uni-freiburg.de/data/8862.

S. Brdar, A. Dedner, and R. Klöfkorn. Compact and Stable Discontinuous Galerkin Methods with Application to Atmospheric Flows. In I. K. et al., editor, *Computational Methods in Science and Engineering: Proceedings of the Workshop SimLabs@KIT*, pages 109–116. KIT Scientific Publishing, 2011a. URL http://dx.doi.org/10.5445/KSP/1000023323.

S. Brdar, A. Dedner, R. Klöfkorn, M. Kränkel, and D. Kröner. Simulation of Geophysical Problems with DUNE-FEM. In E. K. et al., editor, *Computational Science and High Performance Computing IV*, volume 115, pages 93–106. Springer, 2011b. URL http://dx.doi.org/10.1007/978-3-642-17770-5_8.

S. Brdar, A. Dedner, and R. Klöfkorn. Compact and stable Discontinuous Galerkin methods for convection-diffusion problems. *SIAM J. Sci. Comput.*, 34(1):263–282, 2012a. URL http://dx.doi.org/10.1137/100817528.

S. Brdar, A. Dedner, and R. Klöfkorn. CDG Method for Navier-Stokes Equations. In S. Jiang and T. Li, editors, *Hyperbolic Problems - Theory, Numerics and Applications*, pages 320–327. World Scientific Publishing Co Pte Ltd, 2012b.

S. Brdar, M. Baldauf, A. Dedner, and R. Klöfkorn. Comparison of dynamical cores for NWP models: comparison of COSMO and DUNE. *Theoretical and Computational Fluid Dynamics*, 27 (3-4):453–472, 2013. URL http://dx.doi.org/10.1007/s00162-012-0264-z.

A. Burri, A. Dedner, D. Diehl, R. Klöfkorn, and M. Ohlberger. A general object oriented framework for discretizing non-linear evolution equations. In Y. S. et al., editor, *Advances in High Performance Computing and Computational Sciences*, volume 93, pages 69–87. Springer, 2006. URL http://dx.doi.org/10.1007/978-3-540-33844-4_7.

T. A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004. URL http://dx.doi.org/10.1145/992200.992206.

A. Dedner and J. Giesselmann. A posteriori analysis of fully discrete method of lines dg schemes for systems of conservation laws. *arXiv preprint 1510.05430*, 2015. URL http://arxiv.org/abs/1510.05430.

A. Dedner and Klöfkorn. On Efficient Time Stepping using the Discontinuous Galerkin Method for Numerical Weather Prediction. In *Advances in Parallel Computing*, volume 27, pages 627 – 636. Springer, 2016. URL http://dx.doi.org/10.3233/978-1-61499-621-7-627.

A. Dedner and R. Klöfkorn. The compact discontinuous Galerkin method for elliptic problems. In *Finite volumes for complex applications V*, pages 761–776. ISTE, London, 2008.

A. Dedner and R. Klöfkorn. Stabilization for Discontinuous Galerkin Methods Applied to Systems of Conservation Laws. In E. T. et al., editor, *Proc. of the 12th International Conference on Hyperbolic Problems, Proceedings of Symposia in Applied Mathematics 67, Part 1, 253-268*, 2009.

A. Dedner and R. Klöfkorn. A Generic Stabilization Approach for Higher Order Discontinuous Galerkin Methods for Convection Dominated Problems. *J. Sci. Comput.*, 47(3):365–388, 2011. URL http://dx.doi.org/10.1007/s10915-010-9448-0.

A. Dedner, R. Klöfkorn, and D. Kröner. Higher Order Adaptive and Parallel Simulations Including Dynamic Load Balancing with the Software Package DUNE. In W. N. et al., editor, *High Performance Computing in Science and Engineering '09*, pages 229–239. Springer, 2010a. URL http://dx.doi.org/10.1007/978-3-642-04665-0_16.

A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive scientific computing: abstraction principles and the dune-fem module. *Computing*, 90(3–4): 165–196, 2010b. URL http://dx.doi.org/10.1007/s00607-010-0110-3.

A. Dedner, M. Fein, R. Klöfkorn, D. Kröner, D. Lebiedz, J. Siehr, and J. Unger. On the computation of slow manifolds in chemical kinetics via optimization and their use as reduced models in reactive flow systems. In *Proceedings of the 13th International Conference on Numerical Combustion*, 2011a.

A. Dedner, D. Kröner, and N. Shokina. *Computational Science and High Performance Computing IV: The 4th Russian-German Advanced Research Workshop, Freiburg, Germany, 2009*, chapter Adaptive Modelling of Two-Dimensional Shallow Water Flows with Wetting and Drying, pages 1–15. Springer, 2011b. URL http://dx.doi.org/10.1007/978-3-642-17770-5_1.

A. Dedner, M. Fein, R. Klöfkorn, and D. Lebiedz. On the use of chemistry-based slow invariant manifolds in discontinuous Galerkin methods for reactive flows. Technical report, Institute für Numerische Mathematik, Universität Ulm, 2013. URL https://www.uni-ulm.de/fileadmin/website_uni_ulm/mawi2/forschung/preprint-server/2013/1302_dednerfeinkloefkorn.pdf.

A. Dedner, R. Klöfkorn, and M. Kränkel. Continuous Finite-Elements on Non-Conforming Grids Using Discontinuous Galerkin Stabilization. In J. F. et al., editor, *Finite Volumes for Complex Applications VII*, volume 77 of *Springer Proceedings in Mathematics & Statistics*, pages 207–215. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-05684-5_19.

R. Eymard, G. Henry, R. Herbin, F. Hubert, R. Klöfkorn, and G. Manzini. 3D Benchmark on Discretization Schemes for Anisotropic Diffusion Problems on General Grids. In J. Fort, J. Fürst, J. Halama, R. Herbin, and F. Hubert, editors, *Finite Volumes for Complex Applications VI Problems & Perspectives*, volume 4 of *Springer Proceedings in Mathematics*, pages 895–930. Springer Berlin Heidelberg, 2011. URL http://dx.doi.org/10.1007/978-3-642-20671-9_89.

The Feel++ Consortium. *The Feel++ Book*. 2015. URL https://www.gitbook.com/book/feelpp/feelpp-book.

G. J. Gassner. *Discontinuous Galerkin Methods for the Unsteady Compressible Navier-Stokes Equations*. PhD thesis, Universität Stuttgart, 2009. URL http://elib.uni-stuttgart.de/opus/volltexte/2009/3948/.

S. Girke. *Parallel and Efficient Simulation of Atherosclerotic Plaque Formation Using Higher Order Discontinuous Galerkin Schemes*. PhD thesis, University of Münster, 2017.

S. Girke, R. Klöfkorn, and M. Ohlberger. Efficient Parallel Simulation of Atherosclerotic Plaque Formation Using Higher Order Discontinuous Galerkin Schemes. In J. F. et al., editor, *Finite Volumes for Complex Applications VII*, volume 78 of *Springer Proceedings in Mathematics & Statistics*, pages 617–625. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-05591-6_61.

G. Guennebaud, B. Jacob, et al. Eigen v3, 2010. URL http://eigen.tuxfamily.org.

G. Karniadakis and S. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2005. URL http://www.nektar.info/.

R. Klöfkorn. *Numerics for Evolution Equations — A General Interface Based Design Concept*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2009. https://www.freidok.uni-freiburg.de/data/7175.

R. Klöfkorn. Benchmark 3D: The Compact Discontinuous Galerkin 2 Scheme. In J. Fort, J. Fürst, J. Halama, R. Herbin, and F. Hubert, editors, *Finite Volumes for Complex Applications VI Problems & Perspectives*, volume 4 of *Springer Proceedings in Mathematics*, pages 1023–1033. Springer Berlin Heidelberg, 2011. URL http://dx.doi.org/10.1007/978-3-642-20671-9_100.

R. Klöfkorn. Efficient Matrix-Free Implementation of Discontinuous Galerkin Methods for Compressible Flow Problems. In A. H. et al., editor, *Proceedings of the ALGORITMY 2012*, pages 11–21, 2012.

R. Klöfkorn and M. Nolte. Performance pitfalls in the dune grid interface. In A. Dedner, B. Flemisch, and R. Klöfkorn, editors, *Advances in DUNE*, pages 45–58. Springer Berlin Heidelberg, 2012. URL http://dx.doi.org/10.1007/978-3-642-28589-9_4.

R. Klöfkorn and M. Nolte. Solving the Reactive Compressible Navier-Stokes Equations in a Moving Domain. In K. Binder, G. Münster, and M. Kremer, editors, *NIC Symposium 2014 - Proceedings*, volume 47. John von Neumann Institute for Computing Jülich, 2014.

D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193(2):357–397, 2004. URL http://dx.doi.org/10.1016/j.jcp.2003.08.010.

D. Kröner. *Numerical Schemes for Conservation Laws*. Wiley & Teubner, Stuttgart, 1997.

T. Malkmus. *Fluid-Structure-Interaction — Simulation of Non-Newtonian Fluid Interacting with Thin Ellastic Shells*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2017.

K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5): 451–466, 2001.

D. Schuster, S. Brdar, M. Baldauf, A. Dedner, R. Klöfkorn, and D. Kröner. On discontinuous Galerkin approach for atmospheric flow in the mesoscale with and without moisture. *Meteorologische Zeitschrift*, 23(4):449–464, 2014. URL http://dx.doi.org/10.1127/0941-2948/2014/0565.

D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173, Dresden, Germany, 2009.

J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

V. Weaver and J. Dongarra. Can Hardware Performance Counters Produce Expected, Deterministic Results? In *3rd Workshop on Functionality of Hardware Performance Monitoring*, Atlanta, GA, December 4, 2010.

