Archive of Numerical Software

Special Issue 2017: Proceedings of the 3rd Dune User Meeting (2015)

Edited by:
Markus Blatt
Bernd Flemisch
Oliver Sander

UNIVERSITÄTS-
BIBLIOTHEK
HEIDELBERG

Archive of Numerical Software

Special Issue 2017: Proceedings of the 3rd Dune User Meeting (2015)

Edited by Markus Blatt, Bernd Flemisch, and Oliver Sander

1 **Editorial: Proceedings of the 3rd Dune User Meeting**
Markus Blatt / Bernd Flemisch / Oliver Sander

3 **Using DUNE-ACFEM for Non-smooth Minimization of Bounded Variation Functions** // Martin Alkämper / Andreas Langer

21 **The DUNE-FEM-DG module**
Andreas Dedner / Stefan Girke / Robert Klöfkorn / Tobias Malkmus

63 **Asynchronous evaluation within parallel environments of coupled finite and boundary element schemes for the simulation of multiphysics problems** // Andreas Dedner / Alastair J. Radcliffe

95 **The interface for functions in the dune-functions module**
Christian Engwer / Carsten Gräser / Steffen Müthing / Oliver Sander

111 **The DUNE-DPG library for solving PDEs with Discontinuous Petrov–Galerkin finite elements**
Felix Gruber / Angela Klewinghaus / Olga Mula

129 **Using DUNE-FEM for Adaptive Higher Order Discontinuous Galerkin Methods for Two-phase Flow in Porous Media** // Birane Kane

151 **System testing in scientific numerical software frameworks using the example of DUNE** // Dominic Kempf / Timo Koch

169 **FunG – Automatic differentiation for invariant-based modeling**
Lars Lubkoll

193 **Extending DUNE: The dune-xt modules**
Tobias Leibner/ René Milk / Felix Schindler

217 **The Dune FoamGrid implementation for surface and network grids**
Oliver Sander / Timo Koch / Natalie Schröder / Bernd Flemisch

UNIVERSITÄTS-
BIBLIOTHEK
HEIDELBERG

Archive of Numerical Software // Special Issue 2017: Proceedings of the 3rd Dune User Meeting (2015)
Dune, the Distributed and Unified Numerics Environment, has been under continuous development for more than 13 years. Several European institutions participate in this development, and over time, a substantial user community has evolved. In order to establish and foster personal contacts within the community as well as between users and developers, a first Dune User Meeting was held in Stuttgart in 2010, followed by a second one that took place in 2013 in Aachen. In 2015, the third Dune User Meeting was held in Heidelberg from 28th to 29th of September. More than 30 users and developers from five European countries attended, presented Dune-related work and engaged in lively discussions. Ten presentations resulted in contributions to these proceedings.

About the Editors

Markus Blatt is an independent consultant and entrepreneur. As an applied mathematician he helps his customers developing, improving, and employing custom simulation software based upon open source and DUNE in particular.

After completing his PhD in applied mathematics, **Bernd Flemisch** turned to the engineering sciences. He is currently associate professor in the Department of Hydromechanics and Modelling of Hydrosystems at the University of Stuttgart. His research interests encompass computational models for porous media flow, transport and deformation phenomena, model coupling and decoupling as well as advanced discretization and solution techniques.

Oliver Sander is professor for numerical mathematics at the Technische Universität Dresden. He investigates and develops simulation methods for problems in mechanics, like contact and fracture mechanics, and plasticity.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

This work is published under the Creative Commons License 4.0 (CC BY-SA 4.0).

The online version of this publication is freely available on the ebook-platform of the Heidelberg University Library heiBOOKS http://books.ub.uni-heidelberg.de/heibooks (open access).
urn: urn:nbn:de:bsz:16-heibooks-book-280-9
doi: https://doi.org/10.11588/heibooks.280.364

© 2017 by the authors.

Cover Image: Adaptive refinement with element parameterizations leads to non-conforming geometries (see p. 231 in this book). © Oliver Sander. Published under CC BY-SA 2.5.

eISSN: 2197-8263

ISBN 978-3-946531-61-6 (Softcover)
ISBN 978-3-946531-60-9 (PDF)

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 1–2
DOI: 10.11588/ans.2017.1.36872

Editorial: Proceedings of the 3rd Dune User Meeting

Markus Blatt[1], Bernd Flemisch[2], and Oliver Sander[3]

[1]Dr. Markus Blatt HPC-Simulation-Software & Services, Heidelberg, markus@dr-blatt.de
[2]University of Stuttgart, Institute for Modelling Hydraulic and Environmental Systems, bernd.flemisch@iws.uni-stuttgart.de
[3]TU Dresden, Institute for Numerical Mathematics, oliver.sander@tu-dresden.de

Dune, the Distributed and Unified Numerics Environment[1], has been under continuous development for more than 13 years. Several European institutions participate in this development, and over time, a substantial user community has evolved. In order to establish and foster personal contacts within the community as well as between users and developers, a first Dune User Meeting was held in Stuttgart in 2010, followed by a second one that took place in 2013 in Aachen. In 2015, the third Dune User Meeting was held in Heidelberg from 28th to 29th of September. More than 30 users and developers from five European countries attended, presented Dune-related work and engaged in lively discussions. Ten presentations resulted in contributions to these proceedings.

M. Alkämper and A. Langer demonstrate how the dune-acfem module simplifies the use of dune-fem, using for a case study the minimization of total variation functions.

A. Dedner, S. Girke, R. Klöfkorn and T. Malkmus present dune-Fem-DG, a module that implements the Discontinuous Galerkin method for solving a wide range of nonlinear partial differential equations.

A. Dedner and A. Radcliffe introduce a computational toolbox based on Dune and the software package Bem++[2] for the solution of coupled finite and boundary element systems on multi-core computers.

C. Engwer, C. Gräser, S. Müthing and O. Sander introduce the module dune-Functions providing new interfaces for discrete and non-discrete functions, using type erasure for efficient and readable code.

F. Gruber, A. Klewinghaus and O. Mula introduce dune-DPG, a library for solving partial differential equations with discontinuous Petrov–Galerkin finite elements, a modern approach for formulating inf–sup stable discrete variational formulations.

B. Kane presents higher order discontinuous Galerkin discretizations of a two-phase flow model describing subsurface flow in strongly heterogeneous porous media, considering a fully implicit, locally conservative approach on adaptively generated meshes.

[1]dune-project.org
[2]www.bempp.org

D. Kempf and T. Koch describe a collection of tools for system testing of scientific software.

L. Lubkoll presents a highly efficient library for the automatic differentiation of energy functionals from hyperelasticity.

R. Milk, F.T. Schindler and T. Leibner introduce the DUNE-XT library that complements the core DUNE modules by several concepts and utilities that make generic programming using DUNE even more powerful.

O. Sander, T. Koch, N. Schröder and B. Flemisch introduce DUNE-FOAMGRID, a new implementation of the DUNE grid interface for one- and two-dimensional grids in a physical space of arbitrary dimension, allowing for curved domains and network grids.

Acknowledgment We would like to thank all authors and reviewers for writing and evaluating the excellent contributions to this special issue. We would also like to express our gratitude to Guido Kanschat, editor of the Archive of Numerical Software, for his guidance and encouragement.

 © by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 3–19
DOI: 10.11588/ans.2017.1.27475

Using DUNE-ACFEM for Non-smooth Minimization of Bounded Variation Functions

Martin Alkämper[1] and Andreas Langer[1]

[1]Institute for Applied Analysis and Numerical Simulation, University of Stuttgart

Received: January 29th, 2016; **final revision:** October 6th, 2016; **published:** March 6th, 2017.

Abstract: The utility of DUNE-ACFEM is demonstrated to work well for solving a non-smooth minimization problem over bounded variation functions by implementing a primal-dual algorithm. The implementation is based on the simplification provided by DUNE-ACFEM. Moreover, the convergence of the discrete minimizer to the continuous one is shown theoretically.

1 Introduction

The DUNE-framework [7] and in particular the discretization module DUNE-FEM [11] provides the means to handle discrete functions, operators and solvers on different grids. Still, implementing complicated partial differential equations (PDEs) and their solvers is cumbersome and tedious. The DUNE-module DUNE-ACFEM aims to simplify the usage of DUNE-FEM by defining expression templates for discrete functions and PDE models. This allows to linearly combine discrete functions and PDE models. Additionally it supports parallel and adaptive finite-element schemes on continuous discrete functions for the predefined and combined models [14]. The flexibility of DUNE (and DUNE-FEM, DUNE-ACFEM) allows e.g. to exchange discrete spaces by a single line of code or to change the gridtype and linear solvers.

We will demonstrate the ease of implementation with DUNE-ACFEM by minimizing a non-smooth functional consisting of a combined L^1/L^2-data fidelity term and a total variation term. Such an optimization problem has been shown to effectively remove Gaussian and salt-and-pepper noise, see [16, 19]. In order to compute an approximate solution we use the primal-dual algorithm proposed in [10], which requires a saddle point formulation of the problem. For the numerical implementation we discretize using finite-element spaces defined over locally refined conforming grids. Motivated by the works [3, 4, 5, 17], where the considered functional is composed solely of an L^2-data term and a total variation term, we refine the grid adaptively using an a priori criterion. Similar as in [3] we show for the considered minimization problem, that a minimizer over a finite element space converges to a minimizer in the space of functions of bounded variation as the mesh-size goes to 0.

In contrast to previous works [3, 4, 5, 17], we consider an additional non-smooth L^1-data term in the objective, which has to be treated carefully. Moreover, due to the use of DUNE-ALUGRID [1] and the capabilities of DUNE-ACFEM the resulting algorithm is intrinsically parallelized by domain decomposition.

The rest of the paper is structured as follows. In Section 2 we formulate the continuous and the discrete problem with the respective discrete spaces. In particular for a certain discretization we prove that the discrete problem converges to the continuous one as the mesh-size goes to zero. In Section 3 we give a short overview of Dune-ACFem and discuss how it is used to implement the primal-dual algorithm. Numerical examples showing applicability of our proposed implementation and experiments testing different discretizations are presented in Section 4. Finally in Section 5 we conclude with a short summary and possible future research.

2 Problem Formulation

We consider the following problem

$$\min_{v\in BV(\Omega)\cap L^2(\Omega)} J_{\alpha_1,\alpha_2}(v) := \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} + |Dv|(\Omega), \tag{1}$$

where $\Omega\subset\mathbb{R}^d$, $d\in\mathbb{N}$, is an open bounded set with Lipschitz boundary, $g\in L^2(\Omega)$ is a given datum, $\alpha_i\geq 0$ for $i=1,2$ with $\alpha_1+\alpha_2>0$ and $BV(\Omega)\subset L^1(\Omega)$ denotes the space of functions with bounded variation. That is, $v\in BV(\Omega)$ if and only if

$$|Dv|(\Omega) := \sup\left\{\int_\Omega v \operatorname{div}\vec{\phi}dx,\ \vec{\phi}\in C_0^\infty(\Omega,\mathbb{R}^d),\ \|\,|\vec{\phi}|_2\|_{C^0(\Omega)}\leq 1\right\} \tag{2}$$

is finite, see [2, 13]. The space $BV(\Omega)$ endowed with the norm $\|v\|_{BV(\Omega)} = \|v\|_{L^1(\Omega)} + |Dv|(\Omega)$ is a Banach space [13]. For $\alpha_2>0$ the minimization problem (1) admits a unique solution owing to the strict convexity of the quadratic term [19]. Note, that if $\alpha_1=0$ in (1), then we obtain the functional used in [3, 4, 5, 17].

2.1 Discretization

Let $(\mathcal{T}_h)_{h>0}$ be a sequence of shape-regular triangulations of Ω with diameter $h=\max_{T\in\mathcal{T}_h}\operatorname{diam}(T)$ and $\mathcal{S}_h$ be the set of its mesh entities of codimension 1 (i.e. edges for $d=2$). We define the following finite element spaces

$$\mathcal{L}^0(\mathcal{T}_h) = \{q_h\in L^1(\Omega) : q_h\,|_T \text{ is constant for each } T\in\mathcal{T}_h\}$$
$$\mathcal{S}^1(\mathcal{T}_h) = \{v_h\in C(\overline{\Omega}) : v_h\,|_T \text{ is affine for each } T\in\mathcal{T}_h\}.$$

For the vector-valued versions, we write $\mathcal{L}^0(\mathcal{T}_h)^d$ and $\mathcal{S}^1(\mathcal{T}_h)^d$, respectively, and boundary conditions are denoted via a subindex. In particular we use the subindex $_0$ for zero boundary values and the subindex $_N$ for zero boundary values in normal direction, e.g., $\mathcal{S}^1_0(\mathcal{T}_h) := \{v_h\in\mathcal{S}^1(\mathcal{T}_h) : v_h = 0 \text{ on } \partial\Omega\}$ and $\mathcal{L}^0_N(\mathcal{T}_h)^d = \{q_h\in\mathcal{L}^0(\mathcal{T}_h)^d : q_h\cdot\vec{n}=0 \text{ on } \partial\Omega\}$ where $\vec{n}$ is the unit vector in normal direction. The nodal interpolant $\mathcal{I}_h v\in\mathcal{S}^1(\mathcal{T}_h)$ of a function $v\in W^{2,p}$, with $\frac{d}{2}<p\leq\infty$ or $p=1$ if $d=2$, satisfies

$$\|v-\mathcal{I}_h v\|_{L^p(\Omega)} + h\|\nabla(v-\mathcal{I}_h v)\|_{L^p(\Omega)} \leq c_{\mathcal{I}} h^2\|D^2 v\|_{L^p(\Omega)},$$

where $c_{\mathcal{I}}>0$ is a constant independent of h; cf. [8].

We recall, that the space $BV(\Omega)$ is continuously embedded in $L^p(\Omega)$ for $1\leq p\leq\frac{d}{d-1}$, i.e., there is a constant $c_{BV}>0$ such that $\|v\|_{L^p(\Omega)}\leq c_{BV}\|v\|_{BV} = c_{BV}\left(\|v\|_{L^1(\Omega)}+|Dv|(\Omega)\right)$ for any $v\in BV(\Omega)$. For $1\leq p<\frac{d}{d-1}$ this embedding is compact; cf. [2]. Smooth functions are dense in $BV(\Omega)\cap L^p(\Omega)$, $1\leq p<\infty$. In particular, for $v\in BV(\Omega)\cap L^2(\Omega)$ and $\delta>0$ there exists $\varepsilon:=\varepsilon(\delta)>0$ and functions $(v_\varepsilon)_{\varepsilon>0}\subset C^\infty\cap BV(\Omega)\cap L^2(\Omega)$ such that

$$\|\nabla v_\varepsilon\|_{L^1(\Omega)} \leq |Dv|(\Omega) + c_0\delta, \tag{3}$$
$$\|v-v_\varepsilon\|_{L^2(\Omega)}\leq c_1\delta, \qquad \|v-v_\varepsilon\|_{L^1(\Omega)}\leq c_2\delta, \tag{4}$$
$$\|D^2 v_\varepsilon\|_{L^2(\Omega)}\leq\varepsilon^{-2}\|v\|_{L^2(\Omega)}, \qquad \|D^2 v_\varepsilon\|_{L^1(\Omega)}\leq\varepsilon^{-2}\|v\|_{L^1(\Omega)}, \tag{5}$$

 © by the authors, 2017

cf. [3]. These inequalities follow from standard mollifier techniques, see e.g. [13]. Now we are able to show the following convergence result, which follows similar ideas as the proof of [3, Theorem 3.1].

Theorem 2.1 *Let $u_h \in \arg\min_{v\in\mathcal{S}^1(\mathcal{T}_h)} J_{\alpha_1,\alpha_2}(v)$ and $u \in BV(\Omega)\cap L^2(\Omega)$ be a minimizer of the function J_{α_1,α_2}. Then we have that $J_{\alpha_1,\alpha_2}(u_h) \to J_{\alpha_1,\alpha_2}(u)$ as $h \to 0$. If additionally $\alpha_2 > 0$, then $u_h \to u$ in $L^2(\Omega)$ as $h \to 0$.*

Proof *By the optimality of u we have $J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \geq 0$. For $\delta > 0$ let $u_\varepsilon \in C^\infty(\Omega)\cap L^2(\Omega)$ as above and $\mathcal{I}_h u_\varepsilon$ its nodal interpolant, i.e., $\mathcal{I}_h u_\varepsilon \in \mathcal{S}^1(\mathcal{T}_h)$. Then we deduce*

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\leq J_{\alpha_1,\alpha_2}(\mathcal{I}_h u_\varepsilon) - J_{\alpha_1,\alpha_2}(u) \\ &= \|\nabla \mathcal{I}_h u_\varepsilon\|_{L^1(\Omega)} + \alpha_1\|\mathcal{I}_h u_\varepsilon - g\|_{L^1(\Omega)} + \alpha_2\|\mathcal{I}_h u_\varepsilon - g\|^2_{L^2(\Omega)} \\ &\qquad - |Du|(\Omega) - \alpha_1\|u-g\|_{L^1(\Omega)} - \alpha_2\|u-g\|^2_{L^2(\Omega)}. \end{aligned}$$

Using (3) and $\|\mathcal{I}_h u_\varepsilon - g\|^2_{L^2(\Omega)} - \|u-g\|^2_{L^2(\Omega)} = \int_\Omega (\mathcal{I}_h u_\varepsilon - u)(\mathcal{I}_h u_\varepsilon + u - 2g)$ we obtain

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\leq \|\nabla \mathcal{I}_h u_\varepsilon\|_{L^1(\Omega)} - \|\nabla u_\varepsilon\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - g\|_{L^1(\Omega)} \\ &\qquad - \alpha_1\|u-g\|_{L^1(\Omega)} + \alpha_2\int_\Omega (\mathcal{I}_h u_\varepsilon - u)(\mathcal{I}_h u_\varepsilon + u - 2g). \end{aligned}$$

By the triangle-inequality and the Cauchy-Schwarz inequality we get

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\leq \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - g - (u-g)\|_{L^1(\Omega)} \\ &\qquad + \alpha_2\|\mathcal{I}_h u_\varepsilon - u\|_{L^2(\Omega)}(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}) \\ &= \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - u_\varepsilon + u_\varepsilon - u\|_{L^1(\Omega)} \\ &\qquad + \alpha_2\|\mathcal{I}_h u_\varepsilon - u_\varepsilon + u_\varepsilon - u\|_{L^2(\Omega)}(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}) \\ &\leq \|\nabla(\mathcal{I}_h u_\varepsilon - u_\varepsilon)\|_{L^1(\Omega)} + c_0\delta + \alpha_1\|\mathcal{I}_h u_\varepsilon - u_\varepsilon\|_{L^1(\Omega)} + \alpha_1\|u_\varepsilon - u\|_{L^1(\Omega)} \\ &\qquad + \alpha_2(\|\mathcal{I}_h u_\varepsilon - u_\varepsilon\|_{L^2(\Omega)} + \|u_\varepsilon - u\|_{L^2(\Omega)})(\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)}). \end{aligned}$$

The bound $\|\mathcal{I}_h u_\varepsilon\|_{L^2(\Omega)} + \|u\|_{L^2(\Omega)} + 2\|g\|_{L^2(\Omega)} \leq \tilde{c}$, which holds provided that $h \leq \varepsilon$, and the nodal interpolant estimate yield

$$\begin{aligned} J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) &\leq c_{\mathcal{I}} h\|D^2 u_\varepsilon\|_{L^1(\Omega)} + c_0\delta + c_{\mathcal{I}}\alpha_1 h^2\|D^2 u_\varepsilon\|_{L^1(\Omega)} + \alpha_1\|u_\varepsilon - u\|_{L^1(\Omega)} \\ &\qquad + \tilde{c}\alpha_2(c_{\mathcal{I}} h^2\|D^2 u_\varepsilon\|_{L^2(\Omega)} + \|u_\varepsilon - u\|_{L^2(\Omega)}). \end{aligned}$$

Using (4), (5), and the bound $\|u\|_{L^2(\Omega)} \leq \tilde{c}$ we get

$$J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \leq C_1\frac{h}{\varepsilon^2} + C_2\delta + C_3\frac{h^2}{\varepsilon^2} + C_4\delta + C_5\frac{h^2}{\varepsilon^2} + C_6\delta.$$

Let h be sufficiently small, i.e., $h \leq \min\{\delta, \delta\varepsilon^2\}$. Then for $\delta \to 0$ we deduce that $h \to 0$ and hence $J_{\alpha_1,\alpha_2}(u_h) \to J_{\alpha_1,\alpha_2}(u)$.

If $\alpha_2 > 0$, then J_{α_1,α_2} is strictly convex and it follows that

$$J_{\alpha_1,\alpha_2}(u_h) - J_{\alpha_1,\alpha_2}(u) \geq \alpha_2\|u_h - u\|^2_{L^2(\Omega)},$$

cf. [19, Lemma 3.8]. Hence $u_h \to u$ for $h \to 0$. □

By the definition of the total variation (2) an equivalent formulation of (1) reads

$$\min_{v\in BV(\Omega)\cap L^2(\Omega)} \max_{\vec{p}\in C_0^\infty(\Omega,\mathbb{R}^d), |\vec{p}|_2\leq 1} \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} + \int_\Omega v \operatorname{div}\vec{p}. \tag{6}$$

© by the authors, 2017

An analog equivalence exists for finite element spaces [3]. In particular, for $v \in \mathcal{S}^1(\mathcal{T}_h)$ we have that

$$|Dv|(\Omega) = \sup_{\vec{p}\in\mathcal{L}^0_N(\mathcal{T}_h)^d, |\vec{p}|_2\leq 1} \int_\Omega \nabla v \cdot \vec{p}\,\mathrm{d}x \tag{7}$$

leading to

$$\inf_{v\in\mathcal{S}^1(\mathcal{T}_h)} J_{\alpha_1,\alpha_2}(v) = \inf_{v\in\mathcal{S}^1(\mathcal{T}_h)} \sup_{\vec{p}\in\mathcal{L}^0_N(\mathcal{T}_h)^d} \int_\Omega \nabla v\cdot\vec{p}\,\mathrm{d}x + \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} - I_{\mathcal{K}}(\vec{p}),$$

where

$$I_{\mathcal{K}}(\vec{p}) := \begin{cases} 0 & \text{if } \vec{p}\in\mathcal{K} \\ \infty & \text{else} \end{cases}$$

is the indicator function of $\mathcal{K} := \{\vec{p}\in L^1(\Omega)^d \;:\; |\vec{p}|_2 \leq 1\}$. For $v\in\mathcal{L}^0(\mathcal{T}_h)$ we obtain

$$|Dv|(\Omega) = \sup_{(\vec{\beta}_S)_{S\in\mathcal{S}_h}, |\vec{\beta}_S|_2\leq 1} \sum_{S\in\mathcal{S}_h} |S|\vec{\beta}_S[v]_S, \tag{8}$$

where $[v]_S \in \mathbb{R}$ defines the jump across $S\in\mathcal{S}_h$ in normal direction. Hence in this case the discrete problem reads as

$$\inf_{v\in\mathcal{L}^0(\mathcal{T}_h)} J_{\alpha_1,\alpha_2}(v) = \inf_{v\in\mathcal{L}^0(\mathcal{T}_h)} \sup_{\vec{p}\in\mathcal{L}^0_N(\mathcal{S}_h)} \sum_{S\in\mathcal{S}_h} |S|\vec{p}_{|S}[v]_S + \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)} - I_{\mathcal{K}}(\vec{p}).$$

While this formulation is equivalent to the primal formulation, in general we cannot expect convergence to the continuous solution, unless all the jumps are correctly represented by a descendant triangulation. For more details we refer the reader to [3, Section 4].

The primal-dual problem does not have a unique solution in general, even if the primal problem is strictly convex ($\alpha_2 > 0$). This is due to the fact, that the dual problem of (1), even for $\alpha_1 = 0$ and $\alpha_2 > 0$, is only convex but not strictly convex, see for example [15, 18].

2.2 Primal-Dual Algorithm

In the following we discretize using finite element spaces $\mathcal{U}$ (e.g. $\mathcal{S}^1(\mathcal{T}_h)$ or $\mathcal{L}^0(\mathcal{T}_h)$) and $\mathcal{P}$ (e.g. $\mathcal{S}^1_0(\mathcal{T}_h)^d$ or $\mathcal{L}^0_N(\mathcal{T}_h)^d$) for the primal variable u and the dual variable $\vec{p}$, respectively. We denote by $\langle\cdot,\cdot\rangle$ the L^2-inner product or the application of an L^2-functional. Following the ideas of Chambolle and Pock [10, Algorithm 1] we formulate our primal-dual algorithm by identifying $F^* : \mathcal{P}\to\mathbb{R}\cup\{+\infty\}$, $G : \mathcal{U}\to\mathbb{R}\cup\{+\infty\}$, and $K : \mathcal{U}\to\mathcal{P}^*$ with

$$F^*(\vec{p}) = I_{\mathcal{K}}(\vec{p}),\; G(v) = \alpha_1\|v-g\|_{L^1(\Omega)} + \alpha_2\|v-g\|^2_{L^2(\Omega)}, \text{ and } \langle Kv,\vec{p}\rangle = \int_\Omega v \operatorname{div}\vec{p},$$

where we assume that each element in $\mathcal{P}$ has a weak derivative. If this is not the case, $\langle Kv,\vec{p}\rangle$ is to be understood using the identifications suggested by the equations (7) and (8). That is $\langle Kv,\vec{p}\rangle = \int_\Omega \nabla v\cdot\vec{p}\,\mathrm{d}x$ and

$$\langle Kv,\vec{p}\rangle = \int_\Omega v\operatorname{div}\vec{p} = \sum_{T\in\mathcal{T}_h}\int_{\partial T} v\vec{p}\cdot\vec{n} = \sum_{S\in\mathcal{S}_h}[v]_S\vec{n}\cdot\int_S\vec{p} = \sum_{S\in\mathcal{S}_h}|S|\vec{p}_{|S}[v]_S,$$

cf. [3, Lemma 4.1], respectively.

 © by the authors, 2017

In the case that $\nabla v \notin L^2(\Omega)$ for $v \in \mathcal{U}$ or $\operatorname{div} \vec{q} \notin L^2(\Omega)$ for $\vec{q} \in \mathcal{P}$ we use the following identities

$$\begin{aligned} \mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{S}_h): &\qquad \langle \nabla v, \vec{q} \rangle := -\sum_{S \in \mathcal{S}_h} |S| \vec{p}_{|S} [v]_S =: -\langle v, \operatorname{div} \vec{q} \rangle, \\ \mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h): &\qquad \langle \nabla v, \vec{q} \rangle := -\langle v, \operatorname{div} \vec{q} \rangle, \\ \mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h): &\qquad \langle \nabla v, \vec{q} \rangle =: -\langle v, \operatorname{div} \vec{q} \rangle. \end{aligned} \tag{9}$$

We assume, that the datum $g \in L^2(\Omega)$ and the cost parameters α_i are given. Then in our algorithm we initialize $u_0 = \bar{u}_0 \in \mathcal{U}$, e.g., $u_0 \equiv 0$ or $u_0 = \mathbb{P}g$, where $\mathbb{P}: L^2(\Omega) \to \mathcal{U}$ denotes the L^2-projection onto the discrete space $\mathcal{U}$, and $\vec{p}_0 \in \mathcal{P}$ to be the zero function. As parameters we need the step sizes $\sigma, \tau > 0$, the coefficient $\beta = \frac{\tau\alpha_1}{1+2\tau\alpha_2}$, and the overrelaxation parameter $\theta \in [0,1]$.
Then our primal-dual algorithm iterates starting with $k = 0$ as follows:

1. Set $\bar{p} \in \mathcal{P}$ with step size σ as

$$\langle \bar{p}, \vec{q} \rangle = \langle \vec{p}_k + \sigma \nabla \bar{u}_k, \vec{q} \rangle \quad \forall \vec{q} \in \mathcal{P}. \tag{10}$$

   If $\nabla \bar{u}_k \in \mathcal{P}$, then we can use the strong formulation, otherwise we use the identities (9). As the algorithm is derived from formulation (6), we need to guarantee that $|\vec{p}_k|_2 \le 1$ for all k. This is done by the following update

$$\vec{p}_{k+1} = (I + \sigma \partial F^*)^{-1}(\bar{p}) \quad \Leftrightarrow \quad \vec{p}_{k+1}(x) = \frac{\bar{p}(x)}{\max(|\bar{p}(x)|_2, 1)}, \tag{11}$$

   for almost any $x \in \Omega$. Note, that due to the structure of the operator $(I + \sigma \partial F^*)^{-1}$ (see [10] for more details) $\vec{p}_{k+1} \in \mathcal{P} \cap \mathcal{K}$.

2. Update u_k using

$$u_{k+1} = (I + \tau \partial G)^{-1}(u_k + \tau \operatorname{div} \vec{p}_{k+1}) \quad \Leftrightarrow \quad u_{k+1}(x) = \begin{cases} z(x) - \beta & \text{if } z(x) - \beta \ge \mathbb{P}g(x) \\ z(x) + \beta & \text{if } z(x) + \beta \le \mathbb{P}g(x) \\ \mathbb{P}g(x) & \text{else}, \end{cases} \tag{12}$$

   for almost any $x \in \Omega$, where $z \in \mathcal{U}$ is defined via

$$\langle z, v \rangle = \frac{1}{1 + 2\tau\alpha_2} \left(\langle u_k + 2\tau\alpha_2 g, v \rangle + \tau \langle \operatorname{div} \vec{p}_{k+1}, v \rangle \right) \quad \forall v \in \mathcal{U}. \tag{13}$$

   If the $\operatorname{div} \vec{p}_{k+1} \notin L^2(\Omega)$, then we use again the identities of (9).

3. As proposed in [10] we overrelaxate to speed up the algorithm:

$$\bar{u}_{k+1} = u_{k+1} + \theta(u_{k+1} - u_k) \tag{14}$$

4. If the stopping criterion

$$\frac{\|u_{k+1} - u_k\|_{L^2}}{\tau} + \frac{\|\vec{p}_{k+1} - \vec{p}_k\|_{L^2}}{\sigma} < TOL \tag{15}$$

   holds, we terminate the algorithm, otherwise we set $k \to k+1$ and repeat.

Note that in equation (11) with $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$ it is a priori not clear that $\vec{p}_{k+1} \in \mathcal{S}^1_0(\mathcal{T}_h)^d$, as taking the pointwise maximum of two piecewise linear functions results in a piecewise linear function on a finer grid. So in this case we additionally apply the nodal interpolation operator $\mathcal{I}_h : C^0(\Omega)^d \to \mathcal{P}$, which then guarantees $\| |\vec{p}_{k+1}|_2 \|_{L^\infty} \le 1$ and $\vec{p}_{k+1} \in \mathcal{P}$. A similar problem arises for $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$

and equation (12), which we treat analogously. Furthermore note, that testing with the complete test-space $\mathcal{U}$ or $\mathcal{P}$ (eqs. (10), (13)), respectively, is equivalent to an L^2-projection onto the respective space.

Theorem 1 of [10] guarantees convergence of this algorithm to a saddle point, if the according discretization of problem (6) has a saddle point and additionally $\theta = 1$ and

$$\sigma\tau B^2 < 1, \tag{16}$$

where $B := \|\nabla\|_{L^2}$ is the operator norm of the gradient operator on the discrete space $\mathcal{U}$. This norm is bounded for discrete functions and of order $O(1/h_{\min})$, where $h_{\min} := \min_{T\in\mathcal{T}_h} \operatorname{diam}(T)$, which implies that using finer meshes will always result in more steps of the primal-dual algorithm. Even using an adaptive mesh is not beneficial to the stepsize, as B depends on the minimum mesh-width.

3 Implementation Details

The algorithm is implemented in the Dune-project non−smooth−minization . This project is compatible with the 2.4-release and available on the website http://www.ians.uni-stuttgart.de/nmh/downloads. It requires the modules Dune-ACFem and all modules that are required by it, available at gitlab.dune-project.org. The installation works the standard Dune-way. (see Appendix A)

3.1 On Dune-ACFem

Dune-ACFem [14] is a simulation framework based on Dune-Fem [11], which is a discretization framework based on Dune. Dune-Fem provides most generally speaking finite-element spaces on generic grids and all the corresponding utilities to construct a finite-element scheme. Examples are given in the Dune-Fem-Howto. For more information consult [11].

The add-on module Dune-ACFem provides expression templates (and also analytical functions, that can be evaluated on the mesh) for the discrete functions of Dune-Fem and also for models similar but more elaborate to those in the Dune-Fem-Howto. This aims at simplifying the algorithmic formulation of elliptic and parabolic PDEs. The expression templates for discrete functions allow for addition, multiplication with scalars, multiplication with scalar functions, scalarproducts of the components of two functions, and unary expressions like componentwise sin, cos, sqrt, exp.

Dune-ACFem is designed to treat 2nd order elliptic PDEs of the form

$$\begin{aligned}
-\nabla\cdot(A(x,u,\nabla u)\,\nabla u) + \nabla\cdot(b(x,u,\nabla u)\,u) + c(x,u,\nabla u)\,u &= f(x) \quad &&\text{in } \Omega,\\
u &= g_D \quad &&\text{on } \Gamma_D,\\
(A(x,u,\nabla u)\nabla u)\cdot\nu + \alpha(x,u)\,u &= g_N \quad &&\text{on } \Gamma_R,\\
(A(x,u,\nabla u)\nabla u)\cdot\nu &= g_N \quad &&\text{on } \Gamma_N,
\end{aligned} \tag{17}$$

or, in weak formulation,

$$\begin{aligned}
\int_\Omega (A\,\nabla u)\cdot\nabla\phi\,dx + \int_\Omega (\nabla\cdot(b\,u) + c\,u)\,\phi\,dx - \int_\Omega f(x)\,\phi\,dx&\\
+ \int_{\Gamma_R} \alpha\,u\,\phi\,do - \int_{\Gamma_N\cup\Gamma_R} g_N\,\phi\,do &= 0,\\
\langle\Pi, u\rangle &= \langle\Pi, g_D\rangle.
\end{aligned} \tag{18}$$

Possible Dirichlet data is enforced by standard Lagrange test-functions Π. The multiplication with the test-functions ϕ and the integral (quadrature) is provided by Dune-Fem. The other part

 © by the authors, 2017

of the integral has to be provided by the model, which is the central concept of DUNE-ACFEM. A model is basically a tuple of the form

$$\mathcal{M} := (\sigma_E,\ \mu_E,\ \rho_I,\ g_N,\ g_D,\ w_D,\ f, \chi_R,\ \chi_N,\ \chi_D, \Psi)$$

with the following constituents:

Flux	$\sigma_E : \mathbb{R}^d \times \mathbb{R}^m \times \mathbb{R}^{m\times d} \to \mathbb{R}^{m\times d}$,
Source	$\mu_E : \mathbb{R}^d \times \mathbb{R}^m \times \mathbb{R}^{m\times d} \to \mathbb{R}^m$,
Robin-flux	$\rho_I : \mathbb{R}^d \times R^m \to \mathbb{R}^m$,
Robin-indicator	$\chi_R : \partial\Omega \to \{0,1\}$,
Neumann-data	$g_N : \mathbb{R}^d \to \mathbb{R}^m$,
Neumann-indicator	$\chi_N : \partial\Omega \to \{0,1\}$,
Dirichlet-data	$g_D : \mathbb{R}^d \to \mathbb{R}^m$,
Dirichlet-indicator	$\chi_D : \partial\Omega \to \{0,1\}$,
Bulk-forces	$f : \mathbb{R}^d \to \mathbb{R}^m$,
Force-functional	$\Psi \in \mathcal{V}^*$.

Using such a model results in the following weak formulation

$$\begin{aligned}\sum_{E\in\mathcal{T}} \Big(\int_E \Big(\sigma_E(x, U, \nabla U) : \nabla V + \mu_E(x, U, \nabla U) \cdot V - f \cdot V\Big) \\ + \int_{I\in E\cap\partial\Omega} \Big(\Big(\chi_R(x)\, \rho_I(x, U) - \chi_N(x)\, g_N(x)\Big) \cdot V\Big)\Big) - \langle \Psi, V\rangle = 0, \\ \langle \Pi, \chi_D\ U\rangle = \langle \Pi, \chi_D\, g_D\rangle.\end{aligned} \tag{19}$$

This is directly reflected in code, as a model is derived from an interface class that requires exactly the implementation of the above constituents (and the linearization of Flux, Source and Robin-flux for non-linear models). Each model is derived from a default *zero-model*, where all the constituents are set to zero, so only non-zero contributions need to be implemented. DUNE-ACFEM allows forming algebraic expressions from existing models. The generated model-expressions fulfill the model-interface just as "hand-coded" ones. This allows forming of complicated models from a set of convenient pre-built skeleton-models. For a list of predefined models consult [14]. The algebraic expressions include linear combinations of models and multiplying with L^∞-functions.

Other important concepts are the `FemScheme` and the `EllipticOperator`. The `EllipticOperator` applies or assembles the discretization of equation (18). The `FemScheme` then solves the given problem and chooses the necessary linear or non-linear solvers, depending on structural information given by the model. DUNE-ACFEM mostly uses preconditioned iterative solvers like CG or GMRes.

3.2 Image Read-In and Noise Simulation

To read-in images we use the CIMG Library - C++ Template Image Processing Toolkit [21]. The single header file library CIMG is an open source project to provide easy handling and processing of images. It is capable of reading-in standard image formats (e.g. `.png`, `.tif`, `.jpg`) and of adding certain types of noise. Each pixel is accessible by its coordinates utilizing the data type `image`, whose constructor gets the filename containing the image location. Moreover, `image` provides a method to add noise to the data with a given noise level for different noise-types. In particular, Gaussian noise with noise level (variance) γ and mean 0 and salt-and-pepper noise S with noise level $s \in [0,1]$, which corrupts an original image $\hat{g}$ by

$$S\hat{g}(x) = \begin{cases} \min \hat{g} & \text{with probability } s/2 \\ \max \hat{g} & \text{with probability } s/2 \\ \hat{g}(x) & \text{with probability } 1-s \end{cases},$$

are available.

We define two adapter classes within the header file src/datafunction.hh to convert either the data type image or an algebraic expression into a DUNE-FEM discrete function. They derive from the GridFunctionAdapter that creates discrete functions from an evaluation method on a mesh element. These functions are defined on the unit square $[0, 1] \times [0, 1]$, as we can scale any rectangular image accordingly. The image resolution and aspect ratio relate to the level of initial refinement and to the number of cells in each direction. So n uniform refinements with $l \times k$ initial cells lead to an image resolution of size $l2^n \times k2^n$ square pixels, where every square is composed of two triangles and l/k corresponds to the aspect ratio. This is done in the gridfile data/unitsquare2d.dgf which describes the unit square using the DUNE DGF – Interval [6] nomenclature.

The artificial addition of noise (NSM_USE_NOISE=**true**) is only needed for demonstration purposes as in real-world application the image is already corrupted by noise due to certain physical processes. We currently add two independent types of noise, where every noise implemented by CIMG can be applied. Noise type and noise level are controlled by a set of parameters defined in the parameter file. For instance, setting nsm.noiseType1 to 0 leads to Gaussian noise and setting it to 2 corresponds to salt-and-pepper noise. Setting any noise level parameter to 0 disables the respective noise. For other noise types and more information consult the documentation of CIMG .

3.3 Implementation of the Primal-Dual Algorithm

The primal-dual algorithm from Section 2.2 is implemented in the file src/nsm.cc, where some of its subroutines, which depend on the continuity of the discrete spaces $\mathcal{U}$ and $\mathcal{P}$, are outsourced into the file src/phc.hh. The template class ProjectionHelperClass uses partial template specialization to correctly define the methods calculateZ() and entitywiseProjection().

The method phc.entitywiseProjection() realizes equations (10) and (11). Solving eq. (10) is relatively easy, as all steps are provided by DUNE-ACFEM, e.g. for continuous $\mathcal{U}$ and $\mathcal{P}$:

```
void entitywiseProjection ( const ForwardDiscreteFunctionType & uBar, AdjointDiscreteFunctionType
    & p )
  {
    //the gradient model defines the weak gradient using continuous test-functions
    //from the space of p
    auto gradU = gradientModel(uBar);
    auto Dbc0 = dirichletZeroModel(p);
    auto mass = massModel(p);
    //we solve p = sigma * nabla u + p
    //with dirichlet zero boundary
    auto model = mass - sigma_ * gradU - p + Dbc0 ;
    //Testspace = AdjointDiscreteFunctionSpaceType = space of p
    typedef EllipticFemScheme<AdjointDiscreteFunctionType , decltype(model)> SchemeType;
    //p is the returned solution
    SchemeType scheme(p, model);
    scheme.solve();
```

In the above case, for given $\vec{p}$ and $\bar{u}$ the equation

$$0 = \langle \bar{p}, \phi \rangle - \sigma \langle \bar{u}, -\operatorname{div} \phi \rangle - \langle \vec{p}, \phi \rangle \quad \forall \phi \in \mathcal{P}$$

with dirichlet-zero boundary conditions is solved with respect to $\bar{p}$. The solution is written into the variable $\vec{p}$.

Projecting $\vec{p}$ to be feasible (eq. (11)) is not in the features of DUNE-ACFEM. So we have to use the features of DUNE-FEM directly. We choose to do the projection entity-wise, i.e. we iterate over all entities of the grid. On each entity we iterate over the degrees of freedom and if $|\vec{p}|_2 > 1$ holds, we restrict the corresponding value to length 1 in the same direction. If the polynomial order of $\mathcal{P}$ is less or equal than 1, this implies the necessary condition $\||\vec{p}|_2\|_{L^\infty} \leq 1$. For the space $\mathcal{S}^1(\mathcal{T}_h)$ this implies the application of the nodal interpolant $\mathcal{I}_h$.

 © by the authors, 2017

The calculation of z (eq. (13)) translates very nicely into code. We use the L2Projection of DUNE-ACFEM and its ability to linearly combine discrete functions. If $\mathcal{P}$ is continuous, the weak form of eq. (13) is

$$\langle z_k, \psi \rangle = \frac{1}{1 + 2\tau\alpha_2} \langle u_k + \tau \operatorname{div} \vec{p}_{k+1} + 2\tau\alpha_2 g, \psi \rangle \quad \forall \psi \in \mathcal{U}$$

and translates into code in the following way:

```
void calculateZ (const AdjointDiscreteFunctionType &p, const ForwardDiscreteFunctionType & uOld,
    ForwardDiscreteFunctionType& z)
  {
    auto divP  = divergence(p);
    L2Projection(1./(1.+tau_*lambda_2_) * ( tau_ * divP + uOld + tau_ * lambda_2_ * projG_), z);
  }
```

For $\mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$ we use the weak divergence to shift the derivative to the test-space $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$. As above we use the basic Models of DUNE-ACFEM, in particular the massModel and the weakDivergenceModel to implement eq. (13) as follows:

```
void calculateZ (const AdjointDiscreteFunctionType &p, const ForwardDiscreteFunctionType & uOld,
    ForwardDiscreteFunctionType& z)
  {
    //get the mass model of the Forward space
    auto U_Phi = massModel(projG_.space());
    //calculate z
    auto weakDiv_P = weakDivergenceModel(p);
    auto projModel = U_Phi -1./(1.+tau_*lambda_2_) * ( tau_ * weakDiv_P + uOld + tau_ * lambda_2_
        * projG_);
    typedef EllipticFemScheme<ForwardDiscreteFunctionType , decltype(projModel)> SchemeType;
    SchemeType scheme(z, projModel);
    scheme.solve();
  }
```

Note that in contrast to the implementation of the method entitywiseProjection() we construct the massModel from a discrete space instead of a discrete function. This is explicitly allowed by DUNE-ACFEM as the object simply has to provide the data type for the test-function space.

For the case of $\mathcal{P} = \mathcal{L}^0_N(\mathcal{S}_h)^d$ and $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$ the summand weakDiv_p is turned into a functional. This functional is called EdgeWeakDivergenceFunctional and is contained in the file edgeweakdivergencefunctional.hh . The main implementation is done in the coefficients() method, that implements the application of the functional to the basis functions in DUNE-FEM notation.
In the resulting code for the method calculateZ(), (see code example above) the main difference is that weakDivergenceModel(p) is replaced by edgeWeakDivergenceFunctional(p). Additionally weakDiv_p in the definition of projModel has been moved outside the brackets, as functions are not allowed to be added to functionals outside a model (see below).

```
    auto weakDiv_P = edgeWeakDivergenceFunctional(p);
    auto projModel = U_Phi -1./(1.+tau_*lambda_2_) * ( uOld + tau_ * lambda_2_ * projG_) -
        tau_/(1.+tau_*lambda_2_) * weakDiv_p;
```

3.4 Adaptive Refinement

Adaptive refinement is initiated by a positive parameter nsm.localRefine. This parameter denotes the number of additional local refinements to be done in the initialization phase in addition to the uniform refinements. If it is set to ≤ 0, no local refinement will be performed. The grid is locally refined at discontinuities of the piecewise constant datum $g \in \mathcal{L}^0(\mathcal{T}_h)$ in the following way: Given an entity E and its neighbour N, if

$$|g_{|N} - g_{|E}| > ADAPTTOL \tag{20}$$

© by the authors, 2017

holds, then E is marked for refinement, where the value of $ADAPTTOL$ is given by the parameter `nsm.adaptTolerance`. We do not refine during the iterations of the primal-dual algorithm, but keep the mesh static. So the refinement increases the resolution of discontinuities and hence drastically improves the projection $\mathbb{P}g$. This is implemented in the method `adaptGrid()` in the file `nsm.cc`. We use the grid manager DUNE-ALUGRID [1], which is capable of handling the needed conforming, parallel, adaptive, triangular grids.

4 Numerical Experiments

To reproduce the data of the experiments of this section, consult Appendix A and follow the described steps.

4.1 Comparison of Discrete Spaces

We investigate for different discrete spaces $\mathcal{U}$ and $\mathcal{P}$ the behaviour of our algorithm with respect to the non-smooth minimization problem (1) considering the following setups

$$\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d, \qquad \mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d,$$
$$\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d, \qquad \mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{S}_h)^d.$$

The setups using $\mathcal{P} = \mathcal{L}^0_N(\ldots)^d$ are motivated by the equivalence of the mixed formulation and the primal formulation for these discrete settings. For the case $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$ we even have guaranteed convergence to the continuous formulation, see Theorem 2.1. The setting $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$ is motivated by [5], where it is shown that for $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$ the discrete pre-dual of the functional J_{0,α_2} Γ-converges to the continuous one. However, setting $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$ does not seem necessary for the proof there, but can be for example replaced by choosing $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$. Therefore we also investigate the case $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$.

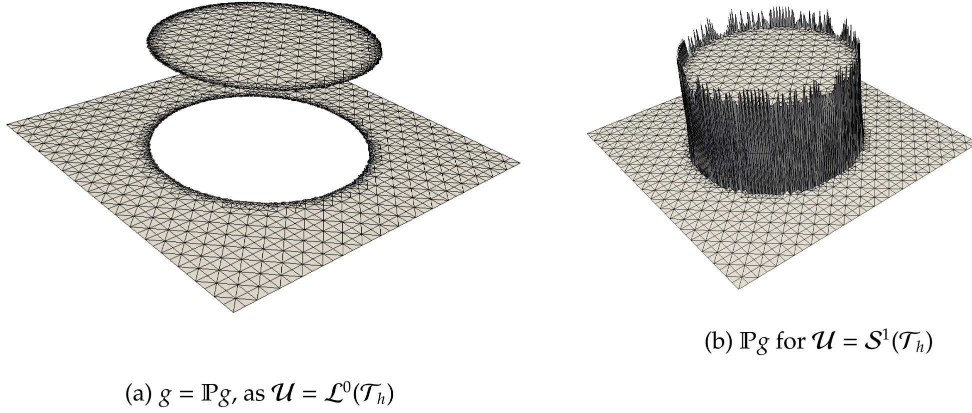
For this set of experiments we choose a similar example as in [5], i.e., the observed data is 1 on a disk of radius 0.3 and 0 elsewhere. Hence, we choose the discrete datum g to be given as a piecewise constant approximation of the function

$$f(x) = \begin{cases} 1, & \text{if } |x - (0.5, 0.5)| \leq 0.3 \\ 0, & \text{else.} \end{cases}$$

Note that this datum changes under grid refinement. The cost parameters are set to $\alpha_1 = 10$ and $\alpha_2 = 20$, the stopping tolerance to 10^{-2}, and the adaptation tolerance to 0.1. For the $\mathcal{L}^0/\mathcal{L}^0$ case we choose the tolerance 10^{-4} as we use an extension of $\mathcal{P}$ into Ω to calculate the part of $\vec{p}$ in the stopping criterion $\frac{\|u_{k+1}-u_k\|_{L^2}}{\tau} + \frac{\|\vec{p}_{k+1}-\vec{p}_k\|_{L^2}}{\sigma} < TOL$. We do $a = 3$ uniform refinements and $b = 5$ additional local refinements. As convergence is guaranteed, if condition (16) holds, and since we know that in general $\|\nabla u\|_{L^2} < Ch\|u\|_{L^2}$ for $u \in \mathcal{U}$, the step sizes are automatically set to $\tau = \sigma = L * 2^{-(a+b)}$ with a constant L. For the $\mathcal{L}^0/\mathcal{L}^0$ case, numerics indicate, that it is possible to even set $\tau = \sigma = L * 2^{-(a+b)/2}$. Note that the choice of L does not only depend on the operator norm of the gradient on the domain, but also on the number of elements in the initial grid. Here we choose $L = 0.19$. The overrelaxation parameter θ is chosen to be 1.

By imposing all equations weakly the primal-dual algorithm de facto uses the L^2-projection $\mathbb{P}g \in \mathcal{U}$ instead of the datum $g \in \mathcal{L}^0(\mathcal{T}_h)$. Figure 1 shows that in the case of $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$ by projecting a discontinuous function onto a continuous space we introduce an additional error in contrast to the case $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$. The over- and undershoots of the continuous projected data do not pose a problem, because they are regularized over the course of the algorithm, as we can see in Figure 2b. Comparing Figure 2a with 2b and Figure 2c with 2d we see that choosing $\mathcal{P} = \mathcal{S}^1(\mathcal{T}_h)^d$ smears out jumps, i.e., the discontinuities are less accurately approximated. This is due to the fact, that $\mathcal{P}$ is continuous and its basis functions have a larger support than if it were

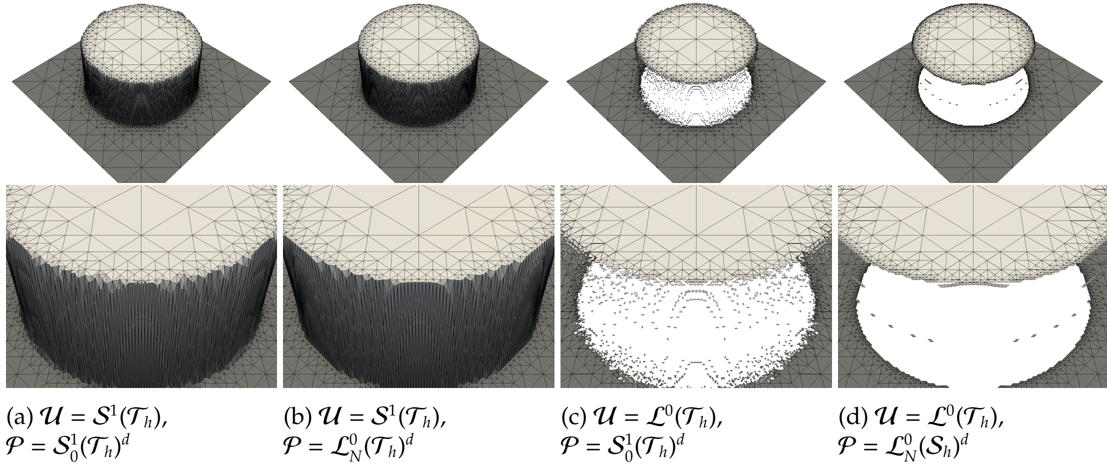
 © by the authors, 2017



(a) $g = \mathbb{P}g$, as $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$

(b) $\mathbb{P}g$ for $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$

Figure 1: The Projection of the piecewise constant datum g onto the space $\mathcal{U}$.



(a) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$

(b) $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$

(c) $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$

(d) $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{L}^0_N(\mathcal{S}_h)^d$

Figure 2: The discrete minima u^* of the functional $J_{10,20}$

discontinuous. Additionally $\mathcal{P} = \mathcal{S}^1(\mathcal{T}_h)^d$ is not the space for which the primal and the mixed formulation are equivalent leading to a worse approximation of $|Dv|(\Omega)$. This explains why the value of the functional in Figures 3 and 4 for these variants is higher than choosing $\mathcal{P} = \mathcal{L}^0_N(\ldots)^d$. In particular this worsens the quality of the solutions obtained with $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h), \mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$, as now multiple elements at the jump are hanging mid-air (Figure 2c), which drastically increases the total variation.

Figure 3 depicts the evolution of the energy J_{α_1,α_2} during the iterations for the considered space pairings. We observe that J_{α_1,α_2} is not monotonically decreasing but stagnates at a minimal value after a certain number of iterations (note that the scale of the number of iterations in Figure 3 is logarithmic). This demonstrates, that in all these settings the algorithm converges to a stationary point of the respective discrete problem. Due to the different combinations of spaces it is clear that the stationary points in general do not coincide and hence the minimal energy is different. The combination $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h), \mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$ yields the best result, as its final energy is the smallest among the considered cases.

We observe that choosing $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$ results in a decreasing energy for $h \to 0$, while for

© by the authors, 2017

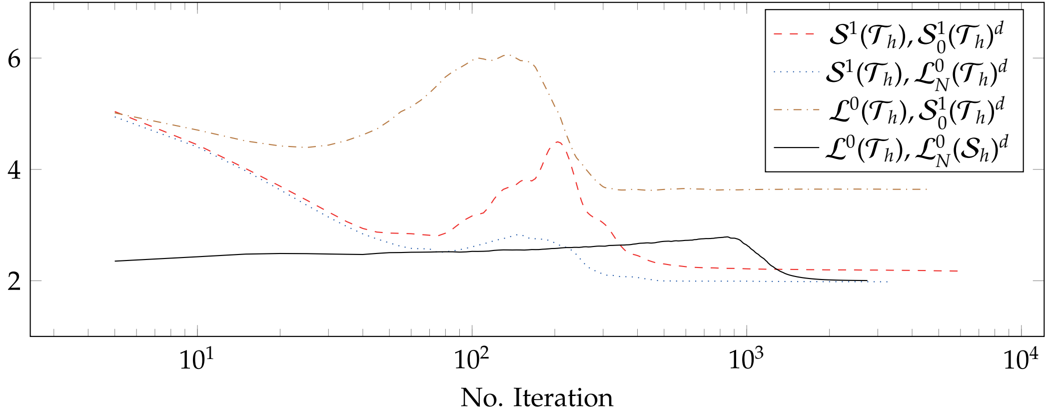


Figure 3: The value of the functional J_{α_1,α_2} over the course of the algorithm.

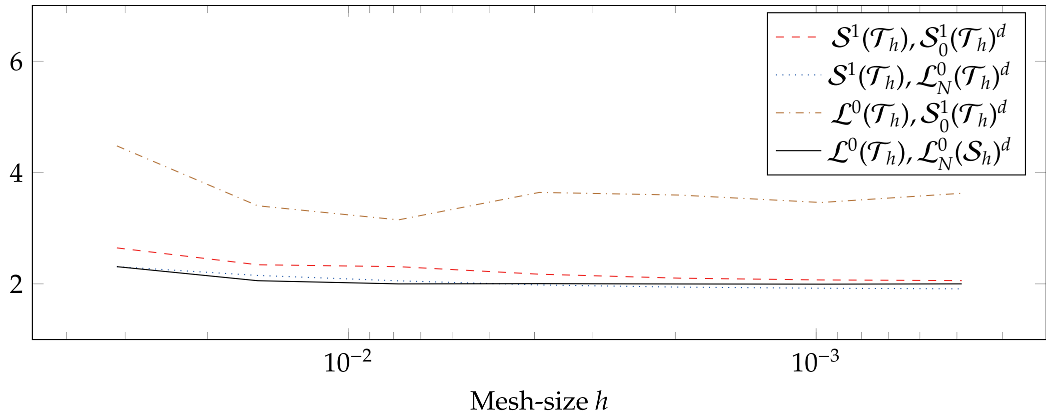


Figure 4: Final energy for $h = 2^{-5}, \dots, 2^{-11}$

$\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$ this is not the case (cf. Figure 4). More precisely for the case $\mathcal{L}^0(\mathcal{T}_h), \mathcal{S}^1_0(\mathcal{T}_h)^d$ the energy seems to oscillate in a certain sense and for $\mathcal{L}^0(\mathcal{T}_h), \mathcal{L}^0_N(\mathcal{S}_h)^d$ the energy stays almost constant. This is due to the fact that, while the approximation of the circle gets better, the total variation does not diminish for $\mathcal{L}^0$. In general approximating functions of bounded variation is not possible with piecewise constant functions on a triangulation, since the length of discontinuities may not diminish over refinement [3]. This is different for the case $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, where discontinuities of functions, that are not representable on the triangulation, can be better approximated.

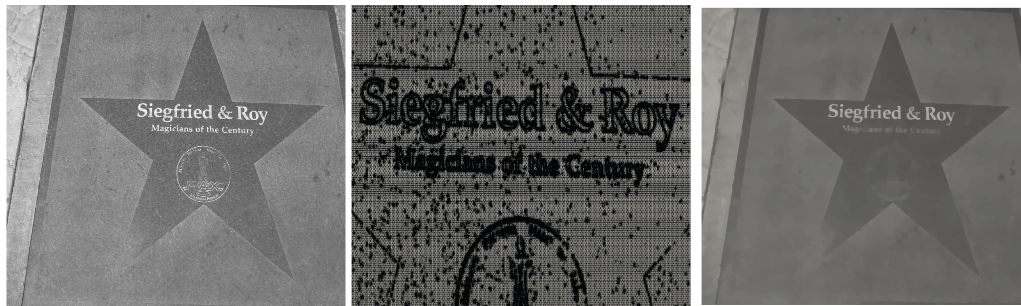
4.2 Image Denoising

Here we demonstrate the denoising capability of the algorithm. Motivated by the above experiments we choose the spaces to be $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$, $\mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$. The considered image with values in $[0,1]$ is first corrupted by Gaussian white noise with variance 0.1 and then salt-and-pepper noise with $s = 0.1$ is added. The obtained image is shown in Figure 5a. In order to reconstruct the image we choose $\alpha_1 = 250$, $\alpha_2 = 150$ in (1), and perform our primal-dual algorithm with the tolerance set to 10^{-2}, $\theta = 1$ and $\sigma = \tau = 2^{-8}$. To resolve the 256×256 pixel-sized picture we apply 8 uniform refinements and no additional local refinement. In Figure 5b we depict the output of our algorithm. The result is reasonably smoothed due to the total variation regularization, while discontinuities are still preserved.

 © by the authors, 2017

(a) Noisy image (b) Result of the algorithm

Figure 5: A corrupted image before and after applying the algorithm



(a) Real image (b) Part of the mesh (c) Result of the algorithm

Figure 6: A real image before and after applying the algorithm

As a second example we run the algorithm on a much finer image, that has a resolution of 2576×1920 pixels (Figure 6a). The spaces are chosen as above, the cost parameters are $\alpha_1 = \alpha_2 = 500$, we run the image on an initial grid of 161×120 squares, each subdivided in two triangles and initiate 2 initial global refinements and 2 local Refinements. So locations with full refinement a triangle is half the size of a pixel. A part of the mesh is depicted in Figure 6b, where one may observe that the discontinuities are captured by the refinement, as intended. The resulting image (Figure 6c) behaves as expected, the noisy structure of the stone is reduced to a minimum, but we also lose some details inside the stars, that we may have wanted to keep. This may be attributed to the choice of the parameters α_1 and α_2. We note that in this example as well as in the previous one, these parameters are chosen at will, but not optimal. For an optimal choice of parameters, we refer the reader to [19]. Moreover it may be of interest to choose local cost parameters, as in [9, 12, 20].

4.3 Strong Scaling

We do a strong scaling experiment on the same datum as in Section 4.1 using the spaces $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$ and $\mathcal{P} = \mathcal{L}^0(\mathcal{T}_h)^d$. The computation is executed on a 32 core shared memory system. The grid manager DUNE-ALUGRID balances the computational load on the initial grid by partitioning it onto the different processors. So we cannot expect the algorithm to scale if the initial mesh is to coarse. E.g. if there are only two initial elements, only two processors can get partitions that are non-empty. Consequently we discretize the unit square by 8×8 elements, see `data/finecube_2d.dgf`. As the grid is already fine, we do not need as many uniform refinements to reach a good resolution, so

we do $a = 3$ uniform and another $b = 3$ local refinements. This results in a different operator norm of the gradient on the initial grid, so we have to set the corresponding parameter nsm.constantL to 0.02 for the stepsizes $\tau = \sigma = L * 2^{-(a+b)}$ to be computed correctly. The tolerance is set to 10^{-2} and $\alpha_1 = \alpha_2 = 150$ to reach a short runtime.

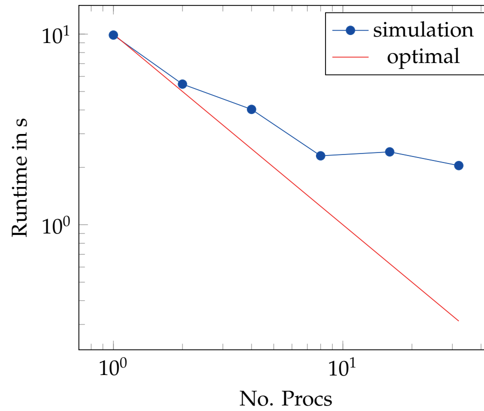


Figure 7: Strong scaling of the algorithm.

Figure 7 indicates that the strong scaling at least up to 8 cores is quite good and afterwards it stagnates, which is probably due to the small grid and not enough workload. We would have to increase the number of initial cells to improve the scaling further. The important part about this result is, that the parallelism is done inside Dune-ACFem and we spent almost no effort to parallelize the code. The only line of code needed initializes MPI.

5 Summary and Outlook

We have shown for a certain discretization that the associated minimizer converges to the continuous one. The primal-dual algorithm used here is implemented very conveniently within Dune-ACFem. The flexibility of Dune-Fem allows us to easily exchange discrete spaces and easily set parameters. Also the algorithm is now intrinsically parallel by domain decomposition with a decent strong scaling. Future research includes implementing a semi-smooth Newton method in Dune-ACFem to increase convergence speed even for highly adaptive grids.

Acknowledgements

The authors would like to thank Claus-Justus Heine for discussions on implementation issues and Dune-ACFem support. Moreover, Martin Alkämper would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart.

Appendix A - Installation Instructions

The program is designed to work in a Linux-environment. It may as well work on a unix machine. To install the program, first download the Dune modules, Dune-common, Dune-geometry, Dune-grid, Dune-istl, Dune-localfunctions, Dune-Fem, Dune-ALUGrid and Dune-ACFem from gitlab.dune-project.org and non−smooth−minization from http://www.ians.uni-stuttgart.de/nmh/downloads. For the Dune- modules checkout the branch releases/2.4. Put all the projects in a directory as direct subfolders. Use your config file config.opts to run the command

```
./dune=common/bin/dunecontrol −−opts=config.opts −−module=non−smooth−minization all
```

 © by the authors, 2017

An example config file example.opts is provided in the main directory of the DUNE project non−smooth−minization .
Now DUNE is installed and the executable nsm from the subdirectory src should have been built within the cmake build directory. The scripts example1.sh to example5.sh produce the results presented in this paper. They require a python interpreter and example3.sh and example5.sh require MPI. The cmake build directory is assumed to be named build−cmake. This can be adjusted by modifying the BUILDDIR variable inside the scripts. The scripts are simple, so changes should be easy. They basically consist of changing directories, compiling the executable nsm, executing it with the right set of parameters and parsing the output into suitable directories. There are two types of output. Console output from std::out is parsed into a file output.graph in a subdirectory of src, which describes the value of the functional and the value of all its parts with respect to both g and $\mathbb{P}g$ and a file output.parameters, where the applied parameters and other output can be looked up. Console output from std::err is displayed on the console. The other type of output is in the cmake build directory located within the directory output in suitable subdirectories. These contain a set of .vtu/.vtk files (readable with Paraview), that contain the datum g (called "image"), the projection $\mathbb{P}g$ (called "projection of g"), the solution (called "u0") and the adjoint variable (componentwise "p0", "p1"). It may take some time to run the scripts. To get faster results, simply edit the LOCREF or INITREF variable of the bash scripts or lower the tolerance parameter nsm.tolerance.

Appendix B - Parameters

There are two types of parameters: run time parameters and compile time parameters.

- Run time parameters can be overloaded on the command line by
  ./nsm nsm.parameter:value ... .
  The default values are set inside the parameter file located at data/parameter. This file is copied into the cmake build directory at compile time. So parameters changed in the build directory will be overwritten when recompiling. Parameters with prefix fem, or istl belong to DUNE-FEM and DUNE-ISTL respectively, and are explained in their documentation. The specific parameters of this algorithm are prefixed nsm. The extensive list of run time parameters reads:

Parameter	Description
nsm.theta	Overrelaxation parameter θ
nsm.maxIt	Maximum number of iterations of the algorithm
nsm.tolerance	The algorithm breaks if the tolerance is reached
nsm.outputStep	Output every nth step
nsm.constantL	If ! NSM_SET_STEPSIZE, $\tau = \sigma = L * 2^{-(a+b)}$
nsm.tau	Stepsize τ
nsm.sigma	Stepsize σ
nsm.lambda_1	L^1-data term cost coefficient α_1
nsm.lambda_2	L^2-data term cost coefficient α_2
nsm.image	Filename of the image to be read in
nsm.initialRefinements	Number of initial uniform refinements a
nsm.localRefine	Number of initial adaptive refinements b
nsm.adaptTolerance	Adaptive tolerance
nsm.noiseType1	The CIMG type of noise to be applied first
nsm.noiseLevel1	The CIMG noise level of the first noise
nsm.noiseType2	The CIMG type of noise to be applied second
nsm.noiseLevel2	The CIMG noise level of the second noise

- Compile time parameters have to be declared when compiling. The location is in the file CMakelists.txt. As they are cmake -cache variables, they can be redefined in the usual way (see e.g. example1.sh). They determine what kind of problem to treat and which discrete

© by the authors, 2017

spaces are to be used.

NSM_SET_STEPSIZE	If true, τ and σ are set manually
NSM_USE_IMAGE	If true, image is used instead of geometric expression
NSM_USE_NOISE	If true, noised imaged is used, requires NSM_USE_IMAGE
NSM_U_DISCONT	If true, $\mathcal{U} = \mathcal{L}^0(\mathcal{T}_h)$, else $\mathcal{U} = \mathcal{S}^1(\mathcal{T}_h)$.
NSM_P_DISCONT	If false, $\mathcal{P} = \mathcal{S}^1_0(\mathcal{T}_h)^d$, else $\mathcal{P} = \mathcal{L}^0_N(\mathcal{T}_h)^d$ or $\mathcal{L}^0_N(\mathcal{S}_h)^d$ (depending on $\mathcal{U}$) .

References

[1] Martin Alkämper, Andreas Dedner, Robert Klöfkorn, and Martin Nolte. The dune-alugrid module. *Archive of Numerical Software*, 4(1):1–28, 2016.

[2] Luigi Ambrosio, Nicola Fusco, and Diego Pallara. *Functions of Bounded Variation and Free Discontinuity Problems*. Oxford Mathematical Monographs. The Clarendon Press, Oxford University Press, New York, 2000.

[3] Sören Bartels. Total variation minimization with finite elements: convergence and iterative solution. *SIAM Journal on Numerical Analysis*, 50(3):1162–1180, 2012.

[4] Sören Bartels. Broken sobolev space iteration for total variation regularized minimization problems. *IMA Journal of Numerical Analysis*, page drv023, 2015.

[5] Sören Bartels. Error control and adaptivity for a variational model problem defined on functions of bounded variation. *Mathematics of Computation*, 84(293):1217–1240, 2015.

[6] 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. II. Implementation and tests in DUNE. *Computing*, 82(2-3):121–138, 2008.

[7] P. Bastian, M. Blatt, A. Dedner, Ch. Engwer, R. Klöfkorn, S.P. Kuttanikkad, M. Ohlberger, and O. Sander. The Distributed and Unified Numerics Environment (DUNE). In *Proc. of the 19th Symposium on Simulation Technique in Hannover, Sep. 12 - 14*, 2006.

[8] Susanne C. Brenner and Ridgway Scott. *The Mathematical Theory of Finite Element Methods*, volume 15. Springer Science & Business Media, 2008.

[9] Chung Van Cao, Juan Carlos De los Reyes, and Carola-Bibiane Schönlieb. Learning optimal spatially-dependent regularization parameters in total variation image restoration. *arXiv preprint arXiv:1603.09155*, 2016.

[10] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, 2011.

[11] 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, 2010.

[12] Yiqiu Dong, Michael Hintermüller, and M. Monserrat Rincon-Camacho. Automated regularization parameter selection in multi-scale total variation models for image restoration. *J. Math. Imaging Vision*, 40(1):82–104, 2011.

[13] Enrico Giusti. *Minimal Surfaces and Functions of Bounded Variation*, volume 80 of *Monographs in Mathematics*. Birkhäuser Verlag, Basel, 1984.

[14] Claus-Justus Heine. Dune-acfem documentation. *http://www.ians.uni-stuttgart.de/nmh/stage/documentation/software/dune-acfem/doxygen/*, 2014.

 © by the authors, 2017

[15] Michael Hintermüller and Karl Kunisch. Total bounded variation regularization as a bilaterally constrained optimization problem. *SIAM Journal on Applied Mathematics*, 64(4):1311–1333, 2004.

[16] Michael Hintermüller and Andreas Langer. Subspace correction methods for a class of nonsmooth and nonadditive convex variational problems with mixed L^1/L^2 data-fidelity in image processing. *SIAM J. Imaging Sci.*, 6(4):2134–2173, 2013.

[17] Michael Hintermüller and Monserrat Rincon-Camacho. An adaptive finite element method in L^2-TV-based image denoising. *Inverse Problems and Imaging*, 8(3):685–711, 2014.

[18] Michael Hintermüller and Georg Stadler. An infeasible primal-dual algorithm for total bounded variation-based inf-convolution-type image restoration. *SIAM J. Sci. Comput.*, 28(1):1–23, 2006.

[19] Andreas Langer. Automated parameter selection in the L^1-L^2-TV model for removing Gaussian plus impulse noise. *accepted by Inverse Problems*, 2016.

[20] Andreas Langer. Automated parameter selection for total variation minimization in image restoration. *Journal of Mathematical Imaging and Vision*, 57(2):239–268, 2017.

[21] David Tschumperlé. The cimg library - c++ template image processing toolkit. *http://www.cimg.eu/reference/.*

© by the authors, 2017

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`

 © by the authors, 2017

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

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

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

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

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.

© by the authors, 2017

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.

 © by the authors, 2017

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

```
// configure macros
#include <config.h>
// include a simulator which is able to call the algorithm creator
#include <dune/fem-dg/misc/simulator.hh>
// include a user defined algorithm creator
#include "algorithmcreator.hh"

int main(int argc, char** argv)
{
  // Initialize MPI (always do this even if you are not using MPI)
  Dune::Fem::MPIManager::initialize(argc, argv);
  try
  {
    // append parameters from command line and remove from argv list
    Dune::Fem::Parameter::append(argc,argv);
    // if parameter file is given append that
    if( argc >= 2)
      Dune::Fem::Parameter::append(argv[1]);
    else
      // read default parameter file
      Dune::Fem::Parameter::append("parameter");

    // select the grid type
    typedef Dune::GridSelector::GridType GridType;
    // define an algorithm via a algorithm creator
    Dune::Fem::MyAlgorithmCreator<GridType> algorithmCreator;

    // run simulation
    Dune::Fem::Simulator::run(algorithmCreator);
  }
  catch(...)
  {
    std::cerr << "Generic exception!" << std::endl;
    return 1;
  }
  return 0;
}
```

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

© by the authors, 2017

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

 © by the authors, 2017

(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

```
1  template <class Algorithm>
2  void compute(Algorithm& algorithm)
3  {
4    // get the grid
5    typedef typename Algorithm::GridType GridType;
6    GridType& grid = algorithm.grid();
7
8    // prepare data output
9    auto dataTup = algorithm.dataTuple();
10   typedef typename Algorithm::DataWriterCallerType::
11     template DataOutput<GridType,decltype(dataTup)>::Type DataOutputType;
12   DataOutputType dataOutput(grid, dataTup);
13
14   // eoc loop
15   for(int eocloop = 0; eocloop < algorithm.eocParams().steps(); ++eocloop)
16   {
17     // call algorithm
18     algorithm.solve(eocloop);
19
20     // write solution to disc
21     dataOutput.writeData(eocloop);
22
23     // refine grid for next eoc step
24     if(algorithm.eocParams().steps() > 1)
25       GlobalRefine::apply(grid,Dune::DGFGridInfo<GridType>::refineStepsForHalf());
26   }
27 }
```

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:

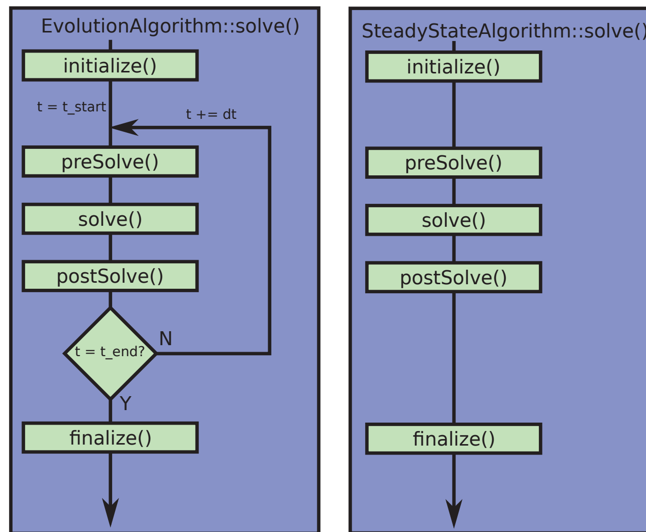
1. `initialize(loop)`, run all the initializing steps which has to be executed once.
2. `preSolve(loop)`, run all the steps preparing the `solve()` step (for instationary algorithms: for each time loop).
3. `solve(loop)`, run a solver.
4. `postSolve(loop)`, run all the steps finalizing the `solve()`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

© by the authors, 2017

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.

 © by the authors, 2017

- 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.

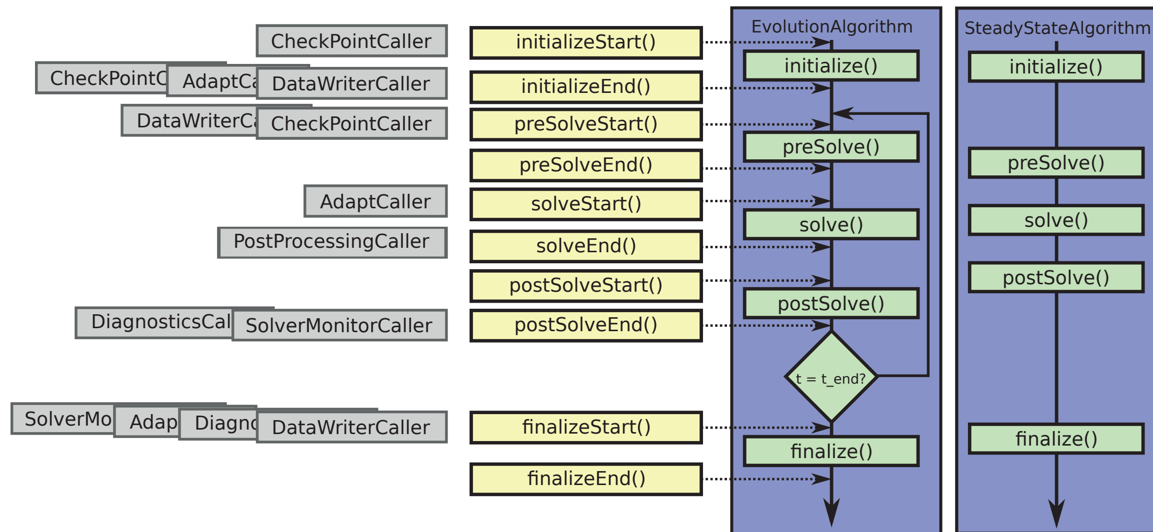
- `DataWriterCaller`, a caller which takes care of data output into different formats.

- `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
```

 © by the authors, 2017

```
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' =====*/
```

 © by the authors, 2017

```
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]),

 © by the authors, 2017

  - `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.

© by the authors, 2017

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 $\mathcal{F}(U)$ or $\mathcal{A}(U, \nabla U)$ 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 $\mathcal{F}(U)$ 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 $\mathcal{A}(U, \nabla U)$ 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.

 © by the authors, 2017

```
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,
```

© by the authors, 2017

```
15                              const double circumEsitimate,
16                              const double time,
17                              const FaceDomainType& x,
18                              const RangeType& u) const;
```

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

C++ code

```
1  // computes velocity in the given evaluation context
2  DomainType velocity(const LocalEvaluationContext& context) const;
3
4  // returns the maximum eigen value of A(U)
5  void eigenValues(const LocalEvaluationContext& context,
6                   const RangeType& u,
7                   RangeType& maxValue) const;
8
9  // evaluate the Jacobian of U (only used by local formulation)
10 void jacobian(const LocalEvaluationContext& context,
11               const RangeType& u,
12               JacobianRangeType& a) const;
13
14 // return penalty factor for diffusive fluxes
15 double penaltyFactor(double time,
16                      const DomainType& x,
17                      const EntityType& entity,
18                      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

```
1 // return true if mass term is non-trivial
2 bool hasMass() const;
3
4 // return mass term
5 inline void mass(const LocalEvaluationContext& context,
6                  const RangeType& u,
7                  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`.

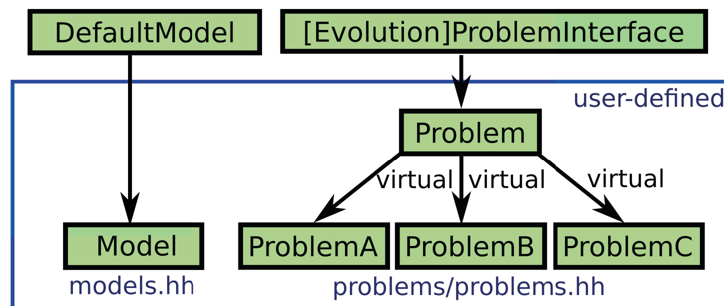
C++ code

```
1   Problem* problem = SubAlgorithmCreator::problem();
2   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.

 © by the authors, 2017

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

```
// getter for the velocity
virtual void velocity(const DomainType& x, DomainType& v) const;
// advection factor
virtual double constantAdvection() const;
// the diffusion matrix
virtual void K(const DomainType& x, DiffusionMatrixType& m) const;
// returns true if diffusion coefficient is constant
virtual bool constantK() const;
// mass factor gamma
virtual double gamma() const;
// the right hand side
virtual void f(const DomainType& x, RangeType& ret) const;
```

Boundary data is set by the following methods:

C++ code

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

For EOC calculation the exact solution has to be known.

C++ code

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

© by the authors, 2017

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

C++ code

```
// latex output for eoc output
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

```
########## DUNE-ISTL ###############
## dune-Istl solvers
istl.preconditioning.method: amg-ilu-0
## number of precondition iterations
istl.preconditioning.iterations: 1
## number of relaxation steps
istl.preconditioning.relaxation: 1

########## PETSc ###################
## PETSc solver: cg, bicg, bicgstab, gmres
petsc.kspsolver.method: cg
## PETSc preconditioning: asm, sor, jacobi, hypre, ml, ilu-n, lu, icc, mumps, superlu
petsc.preconditioning.method: none
## number of precondition iterations
petsc.preconditioning.iterations: 5
```

 © by the authors, 2017

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 $\mathcal{L}_h = \mathcal{L}_{\mathrm{impl},h} + \mathcal{L}_{\mathrm{expl},h}$ into an implicit operator $\mathcal{L}_{\mathrm{impl},h}$ and an explicit operator $\mathcal{L}_{\mathrm{expl},h}$ 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>
```

```
11                                                             ImplicitType;
12 };
13 struct Solver
14 {
15   // defines the linear solver
16   typedef typename AC::template LinearSolvers<DFSpaceType>  LinearSolverType;
17   // defines the ode solver
18   typedef DuneODE::OdeSolverInterface<DiscreteFunctionType> type;
19 };
```

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

```
1  struct Operator
2  {
3    // defines the assembler
4    typedef typename AC::template Operators<MyOpTraits> AssemblerType;
5    // defines a linear operator
6    typedef typename AssemblerType::LinearOperatorType  type;
7  };
8  struct Solver
9  {
10   // defines a linear solver
11   typedef typename AC::template LinearSolvers< DFSpaceType > type;
12 };
```

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.

 © by the authors, 2017

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.

© by the authors, 2017

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.

 © by the authors, 2017

- `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.

 © by the authors, 2017

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].

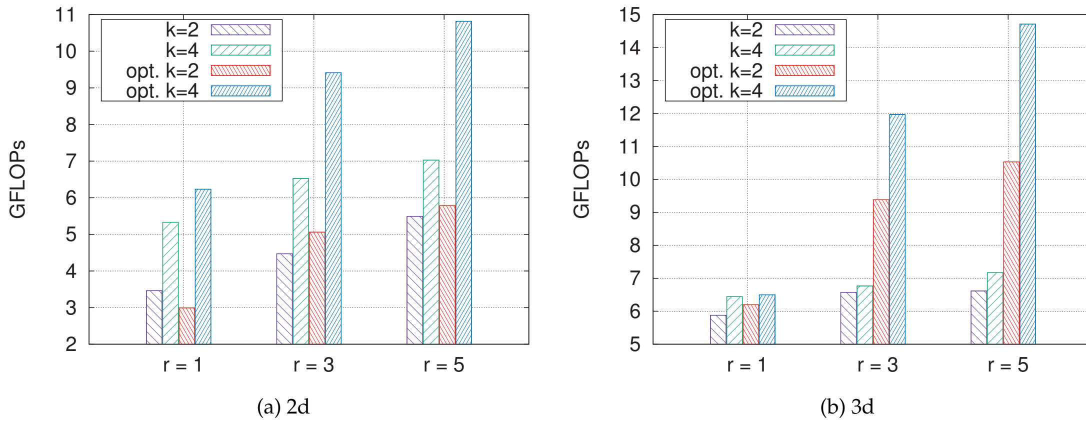


Figure 4: 2d results (left) and 3d results (right) for test case 3 (TC3) using different polynomial orders and different dimensions of the range space.

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

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

© by the authors, 2017

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 $j = 1, \dots, d$:

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

with

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

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

 © by the authors, 2017

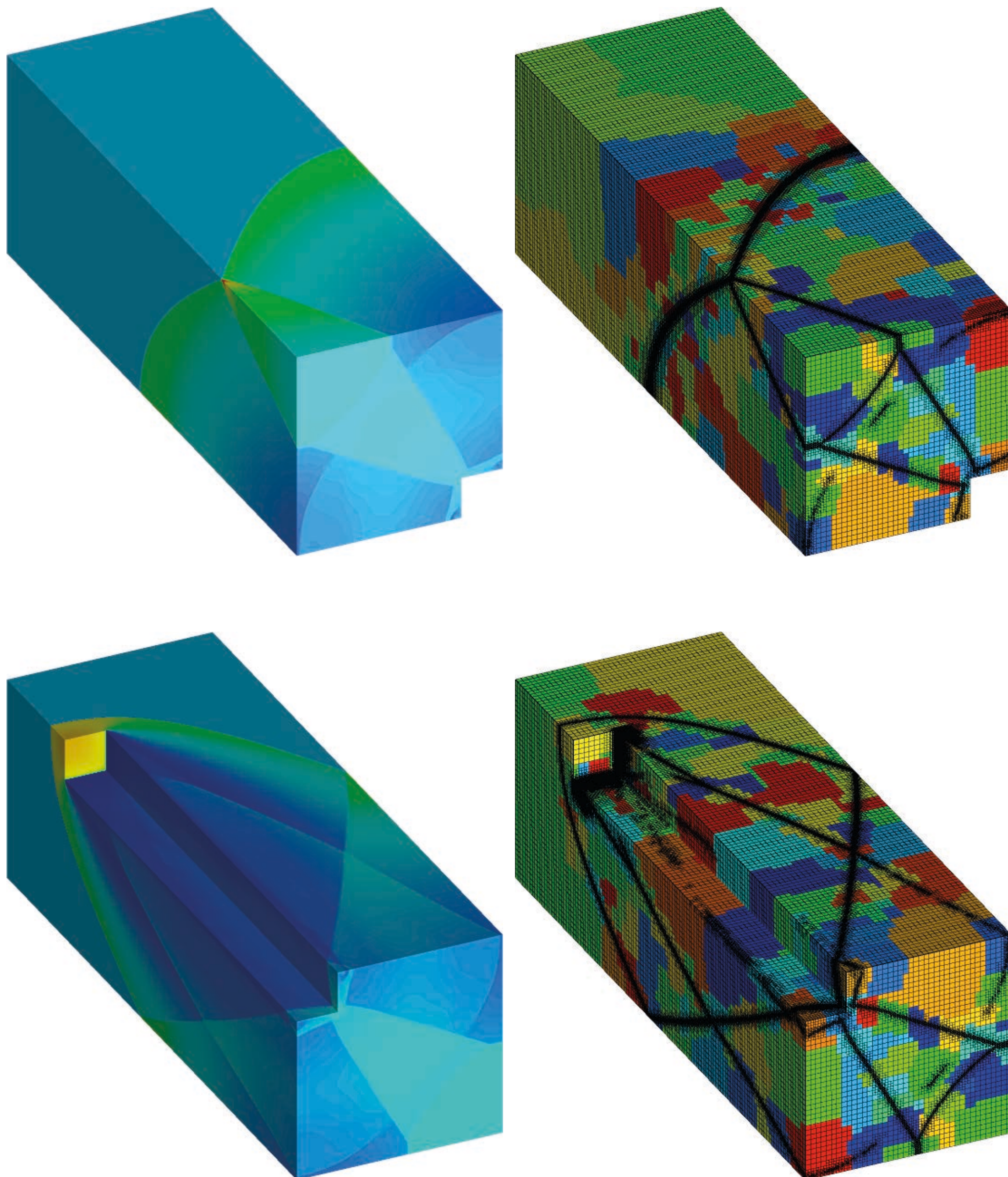


Figure 5: Density distribution obtained with the stabilized DG scheme, adapted grid and partitioning of the grid at time $t = 2$ for test the adaptive Forward Facing Step 3d (see Dedner and Klöfkorn [2011] for details). The calculation used `ALUCubeGrid` and 512 processors. Quadratic basis functions (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	$\|\|U - U_h\|\|_{L^2}$	EOC	$\|\|U - U_h\|\|_{L^2}$	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	$\|\|U - U_h\|\|_{L^2}$	EOC	$\|\|U - U_h\|\|_{DG}$	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 U and 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 $U = (u)^{T}$ be the vector of variables. The Poisson equation is given as follows:

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

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

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

with $K \in \mathbb{R}^{d \times d}$ the diffusion matrix. The source term is $S(U) = f$, 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 $U = (v, p)^{T}$, where p is the pressure and $v = (v_1, ..., v_d)^{T}$ the velocity field. The Stokes equation is given as follows:

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

 © by the authors, 2017

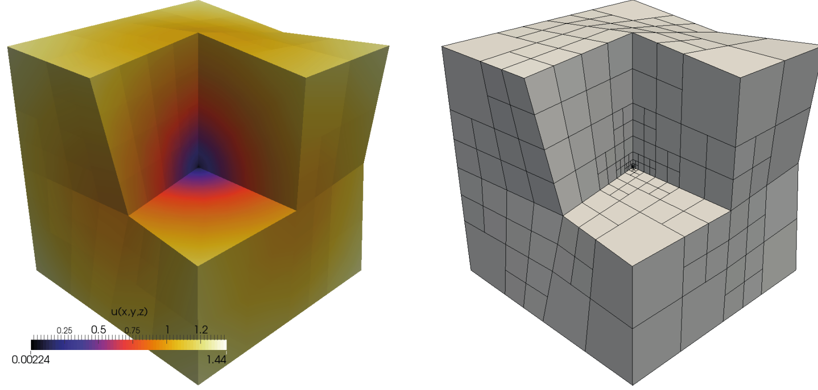


Figure 6: Left, the computed solution u_h at iteration 11 of the adaptive algorithm. Right, the corresponding refined hexahedral grid with non-affine geometry mapping.

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:

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

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

© by the authors, 2017

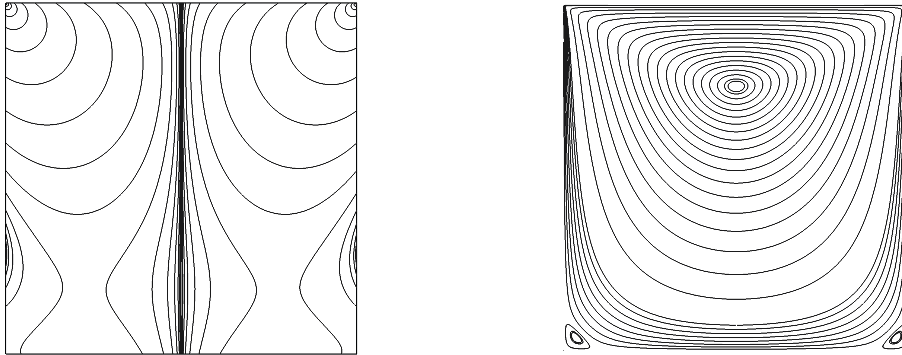


Figure 7: This figure show the numerical solution to the driven cavity test problem. Left: pressure contour lines. Right: stream traces. A huge vortex within the cavity and the two smaller recirculation areas in the lower bottom part can be identified. The simulation was carried out on a structured mesh 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.

 © by the authors, 2017

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.

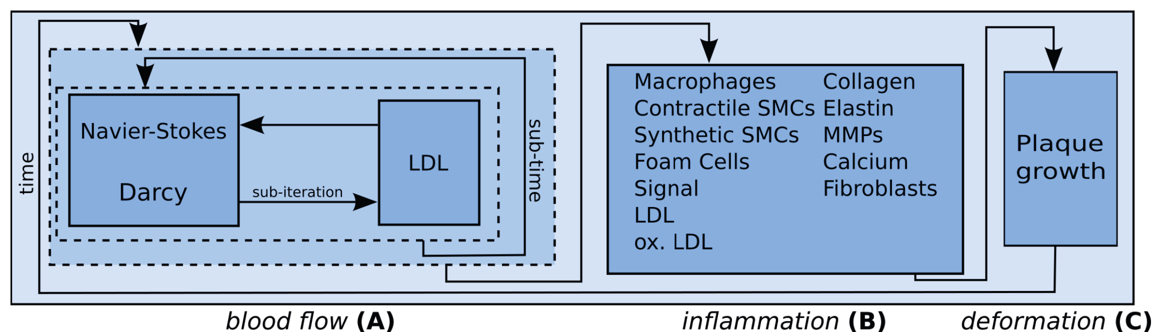
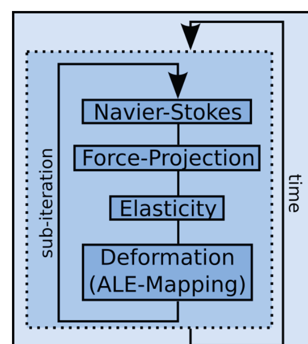


Figure 9: Fluid-Structure Interaction. Different approaches have been developed in the recent years to solve the fluid-structure interaction problem: While one approach is the monolithic one where the whole problem is solved at once, another idea is to use segregated solvers. For each time step a fixed 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

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

After downloading the Dune-Fem-DG module run

Bash code

```
1 ./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

```
1 cd dune-fem-dg/scripts/dune-fem-dg/
2 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.

 © by the authors, 2017

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.

© by the authors, 2017

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

Bash code

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

 © by the authors, 2017

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.

© by the authors, 2017

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.

 © by the authors, 2017

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.

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 63–94
DOI: 10.11588/ans.2017.1.29042

Asynchronous evaluation within parallel environments of coupled finite and boundary element schemes for the simulation of multiphysics problems.

Andreas Dedner[1] and Alastair J. Radcliffe[1]

[1]University of Warwick, Coventry, CV4 7AL, U.K.

Received: March 17th, 2016; **final revision:** October 31st, 2016; **published:** March 6th, 2017.

Abstract: A complete finite element (FEM) and boundary element (BEM) computational toolbox is presented, based on the Dune and Bem++ software packages respectively, for the efficient independent solution on parallel/multi-threaded multi-core computers of separate finite and boundary element systems. Each system has very different memory resource requirements, but can be coupled together within a common computer program for solving multi-physics PDE problems for computational sciences and engineering applications.

Examples of both direct (to a fixed-point) and indirect FEM-BEM iterative coupling, and their performance results with increasing core/thread count, drawn from electromagnetic scattering and fluid mechanics are presented as illustration of the wide scope of applications for this package.

1 Introduction

Finite element methods (FEM) and boundary element methods (BEM) have been around for many years, and due to the fundamental difference in their structures have typically been used for very different types of problem.

For unbounded computational problems, the classic example being an electromagnetic/acoustic wave scattering off an impenetrable obstacle, where a solution is required within a region of space (or even time) of unlimited extent, a finite element discretization of this whole region is impractical.

By requiring just the disretization of the boundaries of computational domains, however, (and hence the name) the boundary element method would avoid such a problem because the boundary of such an infinite region is just the finite surface of the scattering obstacle itself – the “other” boundary to the infinite region can be neglected provided suitable assumptions can be made about the solution at infinity Wrobel [2002].

This capability for supporting a solution in an unmeshed region, however, gives the BEM one of its principle drawbacks, namely that the solutions being “interpolated” in such regions are

assumed to be linear, and this limits its application to a relatively small class of problems where linear (or linearized) solutions are acceptable.

Finite elements, on the other hand, make no such assumptions on a solution, and thus give great flexibility for the simulation of all kinds of non-linear behaviour, but all, of course within a computational region of limited extent.

It should also be noted, that due to the implicitly localized nature of finite element interactions (giving sparse system matrices), parallelization for multi-core processing is relatively easy, with each core just computing the solution for a different physical region of the computational domain.

Because of the "dense" connections between the local values of the unknown solution at different locations in BEM discretizations, however, parallelization of BEM is much more challenging, and usually depends on an overall reduction in the amount of communication between these local values during the solution process through the use of "fast" techniques which can require a huge coding effort to implement.

Now, it is the complimentory, but very different, nature of the finite and boundary element methods which has led to the interest in their coupling together to tackle "mixed" problems Johnson and Nedelec [1980], Stephan [2004], Costabel [1987], Mund and Stephan [1997], with specific applications in electromagnetics Meddahi and Selgas [2003], including the wave scattering suggested above Hiptmair [2003], and fluids/elasticity Brink and Stephan [2001], Gatica and Heuer [2000], Meddahi and Sayas [2000], further references may be found in Radcliffe [2011, 2012].

These coupling schemes have typically involved the combining together of the discretizations afforded by the two methods to form one big system matrix to be inverted, however, the disparate but complementary nature of BEM and FEM means that the structure of this single system matrix can be very non-uniform, making its numerical inversion problematic.

Because the FEM only considers spacially local interations between the discretized values of the solution variables, the system matrices it creates are sparse, and the memory required for the solution of such problems may be distributed analogously to the spacial distribution of the variables themselves in what, following the definitions of Heroux et al. [2011], may be termed a "Mode 1" for the programming of modern multi-node computers with more than one core per node.

The BEM, however, at least in its traditional form, involves dense system matrices arising from the need for all values of the discretized unknown to know about all other values in the boundary integrals upon which the method is based.

It is for this reason that there are sparse and dense regions within the system matrix when FEM and BEM discretizations are combined at the (more fundamental) system matrix level.

The alternative to such a coupling of the system matrices, is to somehow couple the solutions their separate inversions would generate instead. This allows different optimized preconditioners and solvers to be used for the FEM and BEM parts separately.

These coupling methods may be referred to as "iterative", and as will be seen in the next section, involve repeated solution of the separate FEM and BEM problems within each iteration to provide a successive updating, from some initial guess, of the solution and/or its derivative, over the common boundary between the FEM and BEM regions until the updates change nothing and the solution is converged Lin et al. [1996].

Though not used here, these updates can be further defined through the use of relaxation parameters which specify, within a particular iteration, how the solution (or derivative) from one scheme should be used to set the Dirichlet and/or Neumann boundary conditions for the other at the next iteration. By allowing a "mix" of Dirichlet/Neumann data from the two coupled problems to be used as the boundary conditions for either at the next iteration, the convergence of the coupling scheme may be altered significantly.

 © by the authors, 2017

For the examples presented here, though, such mixes are not used, and the data for setting the boundary condition of one problem will come exclusively from the solution of the other problem it is to be coupled to.

Now, while the sparse FEM systems work best on distributed memories, the dense BEM systems work best in shared memory environments, where multiple processes can access all the variable data that they will need over the entire discretized domain to perform their integrals. This involves a "Mode 2" style of programming Heroux et al. [2011], where the parallelization is achieved through the use of multi-threading on a single node with multiple cores.

Extreme-scale computing hardware is typically either specialized towards shared memory designs, with up to 64 cores (beyond which memory clashes become prohibitative), or distributed memory 1000+ cores, with the popular software libraries OpenMP and MPI being often used for the management of each type of memory respectively.

Thus one can say that any dense BEM implementation is well suited for a multithreading environment, for execution on a shared memory machine (as is the case for `Bem++`), while FEM applications are best written using MPI for distributed memory machines (as is the case with the DUNE software packages). Thus a program that were to combine both FEM and BEM solves, would ideally have a hybrid or "bi-modal" memory architecture.

The memory architecture requirements of modern fast, data sparse BEM methods like the H–matrices available in `Bem++` (or other related "fast multipole" type methods) can, however, be significantly different, but will be considered for present purposes as closer to those of their dense forebears, than those of FEM.

As with numerical schemes themselves, the majority of literature on computational data structures has been connected with either the shared or the distributed memory architectures appropriate to the numerical scheme itself, rather than on hybrids. A good discussion on the data-structures and techniques needed to introduce ahared threading libraries into MPI models may be found in Heroux et al. [2011], which also has a few useful references therein.

Recently, approaches for combining the distributed and shared memory paradigms have been investigated – mostly aiming at extending an existing distributed memory model to benefit from the memory hierarchy and hardware structure of modern computer architectures. For example the distribution of the spatial grid for a finite element computation is carried out in two steps, first a coarser distribution over the nodes of the machine using MPI for data communication and in a second step subdividing these coarse patches into smaller chunks distributed over the cores of each node using a shared memory model, see Gaston et al. [2015], Ibanez et al. [2016], Olson et al. [2007] and references therein.

This parallelization strategy is also available in some DUNE modules, for example for matrix free methods in Klöfkorn [2012]. This approach can be thought of as a *hierarchic hybrid* parallelization (or "hierarchic bimodal" to follow the "mode" terminology of Heroux et al. [2011]). It is restricted to single packages and does not require any global access to the whole set of degrees of freedom. This is an important difference to the challenges discussed in the following, where the `Bem++` library requires the whole grid to be available in a shared memory address space, while the DUNE library and the linear solver packages used for solving the FEM problem rely on MPI for data exchange and are not even necessarily thread safe. We are not aware of any openly available publications in the area of computational PDEs employing a *non-hierarchic* hybrid parallelization approach to the use of distributed/shared memory concepts within the same program, and would hope that the present work might be among the first to openly address this issue.

This paper will present the internal structure and design decisions taken for a software toolbox that intertwines the asynchronous evaluation (across a parallel/multithreaded environment) of fem and bem constituent numerical kernels/solvers with disparate but complementary computing resource requirements for the development of efficient integration algorithms for the simulation of coupled partial differential equations arising in multiphysics applications ranging from electromagnetic wave scattering to electrically charged droplet deformations where the underlying

© by the authors, 2017

processes have disparate computing resource requirements that can be exploited effectively in this way.

In particular, we show the construction and management of special shared memory spaces, on a single node, that allow the running of a multi-threaded bem solver within an MPI application using all the cores of the node in which the shared memory sits. The management essentially involves granting ownership of different regions of the shared memory to different processes, or "ranks", to avoid access clashes if two different ranks try to write to the same memory address at the same time. Multiple reading from the same address does not pose the same problems.

Finally, this paper is intended as an introduction and guide to the associated software for use in new hybrid FEM/BEM application developments. While limited investigation and discussion of computational efficiencies is provided, this is from a very experimental "try it and see" approach comprehensible to the largest possible audience, and no formal symbolic analysis of the coupling methods used is presented. However, the reader is directed to Sayas [2009, 2013] and the references therein for a formal analysis of coupling methods similar (or identical) to those used here.

2 Coupling Methods

Consider an equation for an unknown u, valid inside an interior domain, $\Omega \subset \mathbb{R}^3$, satisfying

$$A\,u = f - Bv \tag{1}$$

arising from the discretization of a partial differential equation by the finite element method. Similarly, let the function v be a discrete solution defined on the boundary, $\Gamma \equiv \partial\Omega \subset \mathbb{R}^3$, satisfying

$$D\,v = g - Cu \tag{2}$$

arising again from a discretized partial differential equation using either finite or boundary elements (or both if this surface solution may itself be decomposed into two distinct surface solutions). Any BEM solution would of course be equivalent to solving the corresponding volume problem in the exterior $\Omega^\infty \equiv \mathbb{R}^3 \backslash \Omega$.

The full coupled problem can be written as a (block) matrix vector problem, where A, D arise from a discretized weak formulation of partial differential equations defining a function in the domain Ω and on Γ, respectively. The terms B, C provide coupling between the problem in the domain and on the boundary. The term B provides, for example, Neumann information using the surface solution as flux and C might take Dirichlet or Neumann traces from the domain solution to force the surface problem. Note that the terms A, B, D, C could refer to non-linear operators although we will mostly be focusing on linear operators in the following.

The combined bulk-surface problem thus results in the matrix system

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{3}$$

For the coupling problems discussed here, it is possible to combine the system matrices from both to form a large matrix as shown above. In the case of FEM and BEM coupling this larger matrix will have both sparse and dense regions corresponding to the two different methods used in its construction. Consequently it is crucial for an efficient solution of the coupled system to not treat it as a single matrix, but to make use of its block structure. Furthermore, the methods discussed here allow us to treat the solvers for the different parts of the system as "black boxes", allowing us to use optimal solvers for both the surface and the bulk part of the problem. Thus we can use the efficient methods available in DUNE for solving FEM problems (including multigrid and direct solvers) and the optimal methods for the BEM problems available in `Bem++` (including

 © by the authors, 2017

multipole and H-matrix methods). An efficient solution method is based, for example, on a block-preconditioned GMRes iterative solver Feischl et al. [2015]. We discuss a similar approach below. First we describe the idea based on a simple iterative updating of each scheme's solution allowing convergence to a fixed point solution.

Assume in the following that the matrices on the diagonal result from a bounded, positive bilinear form so that A and D are invertible, and system (3) may be preconditioned by the block diagonal matrix

$$P = \begin{pmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \tag{4}$$

to give the equivalent system

$$\begin{pmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} \tag{5}$$

which may be simplified to

$$\begin{pmatrix} I & A^{-1}B \\ D^{-1}C & I \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} A^{-1}f \\ D^{-1}g \end{pmatrix} \tag{6}$$

Now, one of the simplest iterative schemes one can devise for solving $A\,\mathbf{x} = \mathbf{b}$ is that of Richardson with a relaxation parameter of unity

$$\mathbf{x}^{k+1} = \mathbf{x}^k - A\,\mathbf{x}^k + \mathbf{b} \tag{7}$$

Applying this to (6) with $\mathbf{x} = (u, v)^T$ gives

$$\begin{aligned} \begin{pmatrix} u^{k+1} \\ v^{k+1} \end{pmatrix} &= \begin{pmatrix} u^k \\ v^k \end{pmatrix} - \begin{pmatrix} I & A^{-1}B \\ D^{-1}C & I \end{pmatrix} \begin{pmatrix} u^k \\ v^k \end{pmatrix} + \begin{pmatrix} A^{-1}f \\ D^{-1}g \end{pmatrix} && (8) \\ &= \begin{pmatrix} u^k - u^k - A^{-1}Bv^k \\ v^k - D^{-1}Cu^k - v^k \end{pmatrix} + \begin{pmatrix} A^{-1}f \\ D^{-1}g \end{pmatrix} && (9) \\ &= \begin{pmatrix} A^{-1}\left(f - Bv^k\right) \\ D^{-1}\left(g - Cu^k\right) \end{pmatrix} && (10) \end{aligned}$$

and our first, block Jacobi style iterative coupling solver, to be initiated from an initial guess u^0, v^0 and computed for integer $k > 0$:

$$v^{k+1} = D^{-1}(g - Cu^k)\,, \qquad u^{k+1} = A^{-1}(f - Bv^k) \tag{11}$$

A very simple variant of this gives a second, block Gauss-Seidel style method to solve the coupled problem:

$$v^{k+1} = D^{-1}(g - Cu^k)\,, \qquad u^{k+1} = A^{-1}(f - Bv^{k+1}) \tag{12}$$

It should be noted that very simple Richardson iterative solvers like these were considered in the early days of domain decompositon methods and may have considerable limitations regarding their efficiencies and even convergence, which could be critical or slow for some scenarios Toselli and Widlund [2005].

© by the authors, 2017

The preconditioner P in equation (4) used in the two simple iterative schemes can be modified by using suitable preconditioners instead of the exact inverse matrices. So for example A^{-1} can be replaced by an algebraic multigrid preconditioner as available in DUNE-ISTL. If a Krylov type method is used to invert A then it is possible to fine-tune stopping criteria in each step of the Richardson iteration to improve efficiency of each step, but at the cost of perhaps a slight increase in the number of iterations. This type of efficiency tuning has not been investigated here but could be added to the package.

The final approach discussed here is more involved, but more robust, and often leads to a reduction in the number of iterations required. It makes use of a GMRes method and a reformulation of problem (3) eliminating either v or u: First the second row of (3) gives

$$v = D^{-1}g - D^{-1}Cu \tag{13}$$

which may be substituted into the first row of (3) to give

$$Au + BD^{-1}g - BD^{-1}Cu = f \tag{14}$$

Multiplying through by A^{-1} we have

$$u - A^{-1}BD^{-1}Cu = A^{-1}f - A^{-1}BD^{-1}g \tag{15}$$

Thus solving the coupled system

$$M\,u = b \tag{16}$$

where

$$M = I - A^{-1}BD^{-1}C \tag{17}$$

$$b = A^{-1}\left(f - BD^{-1}g\right) \tag{18}$$

is then completely equivalent to solving system (3). The problem can now be solved using a Krylov iterative solver, for example GMRes. These methods only require computing the matrix vector product $M\,u$.

Of course, it would also have been possible to perform the eliminations the other way around to obtain a combined system $M\,v = b$; solving for the surface variable v instead, with possibly different memory requirements. A third option would be to apply the Krylov method directly to the full block system 6. The framework presented here is flexible enough to allow for an easy implementation of these alternative coupling methods.

Note that the first two coupling methods are included primarily as a simple demonstration on how to implement coupling schemes within the current framework. These methods are well known not to be very robust and efficient but are included to show the appropriate code structure and, due to their simplicity, are intended as a starting point for user developments of coupling schemes/solvers more suited to their particular problems.

Indeed, the use of an appropriate coupling scheme can be crucial to the success of the FEM–BEM coupling as a whole, thus, for the scattering wave case seen in section 6.2, for example, it is suggested the reader try the Jacobi and Gauss-Seidel methods (which do not converge at all without additional relaxation), before restoring the original GMRes coupling scheme more suited to the problem. In the simpler FEM-FEM coupling example found in section 6.1, on the other hand, these two methods *will* converge within a reasonable number of iterations.

 © by the authors, 2017

3 Software Dependencies

The software, version 2.3, to be presented here for the use, and possibly coupling, of FEM and BEM within the same parallel program is based on the DUNE 2.3 and Bem++ software packages, which we now examine in a little more detail.

DUNE stands for the "Distributed and Unified Numerics Environment", and is a modular toolbox for solving partial differential equations (PDEs) with grid-based methods Dune [2012]. It supports the easy implementation of methods like finite elements (FE), and finite volumes (FV).

DUNE is free software licensed under the GPL (version 2) with a runtime exception, thus it is possible to use DUNE even in proprietary software.

The underlying idea of DUNE is to create slim interfaces allowing an efficient use of legacy and/or new libraries. Modern C++ programming techniques enable very different implementations of the same concept (e.g: grids, solvers, ...) using a common interface at a very low overhead. Thus DUNE ensures efficiency in scientific computations and supports high-performance computing applications Dune [2012].

Dune applications are designed and written to use pre-existing packages, or "modules", which can be bolted together to provide whatever functionality the developer will need in their own program (also written as a module). This allows easy sharing of computer code with a "black-box" mentality that anyone using a module should not need to know how it actually works.

Essential to most DUNE programs are the five core modules, DUNE-COMMON, DUNE-GEOMETRY, DUNE-GRID and DUNE-ISTL, which provide the basic housekeeping required of any program handling meshes.

Based on these core modules, the DUNE-FEM module provides all the basis function definitions, matrix assembly and problem definition routines required for the creation of a complete finite element program. If an appropriate discretization module, or "grid-manager", is also used (see next) then parallel assembly and solution of the resulting problems is easily possible for the novice user, with minimal knowledge. The parallelization is based on calls to routines drawn from the Message Passing Interface (MPI) libraries, with a good computational scaling dune-fem howto [2015].

To enable the efficient parallelization of the FEM part of the coupled FEM-BEM solver, the grid manager chosen is DUNE-ALUGRID Alkaemper et al. [2016], which allows the construction and partitioning across multiple processors of any three-dimensional volume and surface meshes the user might specify for the separate FEM and BEM sub-problems. Importantly, it also allows a user defined specification of the partitioning, so that optimal division of the given meshes is possible, with the same partitioning being used for both volume and surface meshes where required to facilitate data exchange.

The last of the DUNE modules required may be thought of as almost literally sticking the whole thing together. With FEM and BEM solutions required on both volume and surface meshes in the various examples that will be presented, it is important to have a means of communicating solution data from one computational domain to another in order that it may be used to update the respective solutions there. The DUNE-GRID-GLUE module Bastian et al. [2010], provides just the simple yet effective means required for transferring various numerical values between the grids (even partitioned and non-matching grids).

Outside of the DUNE environment, and to provide the necessary boundary element routines the Bem++ library is used; an open-source Galerkin boundary element library that handles Laplace, Helmholtz and Maxwell problems on bounded and unbounded domains in three space dimensions Smigaj et al. [2015]. Owing to build compatibility issues, at present the installation script, see appendix §B, downloads a certain git hash of Bem++ which resolves to the 3.0.3 tag. When the Bem++ compatible with DUNE 2.4 becomes available, this restriction is intended to be lifted, with a 2.4 release of the DUNE-FEM-BEM-TOOLBOX.

The package BEM++ is itself built extensively using routines drawn from the `Eigen` C++ template library for linear algebra with matrices, vectors, numerical solvers and related algorithms Guennebaud et al. [2010]. Eigen is a template library defined in the headers and doesn't have any dependencies other than the C++ standard library. Parallelization in `Bem++` is achieved using multi-threading based on Intel's Threading Building Block TBB [2014]. For the grid data structure `Bem++` relies on the Dune core modules, so data exchange between the two packages is achievable without any reorganization of data and thus is highly efficient.

4 Implementing the Coupled Problem

The implementation discussed in this paper is based on the approach adopted in the "howto" module of Dune-Fem dune-fem howto [2015]: both the bulk and the surface problems use a `Model` class describing the functions required for the underlying partial differential equations. This is passed to a `Scheme` class which sets up the required types for the discrete spaces, creates the glue objects that will be needed for any coupling and selects the solvers to be used in addition to also holding the actual solution.

The creation of the glue objects is done automatically by the scheme constructors, with every second scheme defined (within the `main.cc` program file) creating a glue object between its mesh and that of the previously declared scheme. Thus, at least for the present software release, minor attention needs to be paid to the order in which the schemes are declared.

In its original form within Dune-Fem-Howto, the `Scheme` class implements two methods `prepare` and `solve`, which setup the right hand side vector and assembles and solves the system, respectively.

For the Dune-Fem-Bem-Toolbox module the original scheme class used in the Dune-Fem-Howto examples is extended by the additional method `couple` which processes the data communication between two schemes – one "bulk" scheme for the inside region, and one "surface" scheme for the surface of that region.

Thus, in our module we provide two such scheme classes, one for FEM discretized elliptic problems defined either on a surface mesh or on a volume mesh, and a second for exterior BEM problems defined on a surface grid only. In the case of the BEM problem the `Model` only provides the static information about which integral operators to use for the BEM problem. Additional parameters, for example wave numbers, can be set through run time parameters as described below.

For FEM problems systems of non-linear equations of the form

$$\int_{\omega} D(x, u(x, \nabla_{\omega} u(x)) \cdot \nabla_{\omega} \varphi(x) + m(x, u(x), \nabla_{\omega} u(x)) \varphi(x) = \int_{\omega} f(x) \varphi(x)$$

can be defined through the `Model` class. Here $\omega = \Omega$ or $\omega = \Gamma$ is the volume or the surface with $\nabla_\Omega = \nabla$ and ∇_Γ denoting the surface gradient. Although non-linear elliptic problems can be described by the model class and handled by the schemes we will mainly focus on linear problems in the following.

In this version of the module only a linear coupling between the two schemes is considered. This is to be extended in future releases. The surface solution is assumed to be coupled to the volume problem using Dirichlet or Neumann conditions taking the form

$$\nabla u \cdot n + \alpha_\Omega u = \alpha_\Gamma v \, .$$

Here α_Ω and α_Γ are functions defined on the coupling surface.

Note that for a surface FEM problem v are the values of the quantities defined on the surface while for a BEM problem v will actually be an approximation of the normal derivative of the far

 © by the authors, 2017

field solution. Since the $\alpha_\Omega u$ contribution of the coupling will be part of the matrix A, the actual coupling matrix is

$$< Bv, \varphi > := \int_\Gamma \alpha_\Gamma v \varphi \tag{19}$$

The main part of the bulk problem is of the form

$$< Au, \varphi > := \int_\Omega D(x, u, \nabla u) \cdot \nabla \varphi + m(x, u, \nabla u)\varphi + \int_\Gamma \alpha_\Omega u \varphi \tag{20}$$

and described by a single `Model` class consisting of four methods:

C++ code

```
template< class Entity, class Point >
void source ( const Entity &entity, const Point &hatx,
              const RangeType &value, const JacobianRangeType &gradient,
              RangeType &flux ) const
template< class Entity, class Point >
void linSource ( const RangeType& uBar,
                 const Entity &entity, const Point &hatx,
                 const RangeType &value, const JacobianRangeType &gradient,
                 RangeType &flux ) const
template< class Entity, class Point >
void diffusiveFlux ( const Entity &entity, const Point &hatx,
                     const RangeType &value, const JacobianRangeType &gradient,
                     JacobianRangeType &flux ) const
template< class Entity, class Point >
void linDiffusiveFlux ( const RangeType& uBar, const JacobianRangeType& gradientBar,
                        const Entity &entity, const Point &hatx,
                        const RangeType &value, const JacobianRangeType &gradient,
                        JacobianRangeType &flux ) const
```

The methods provide the implementation of $m(x, u, \nabla u), D(x, u, \nabla u)$ and their linearization around $\bar{u}$ on an entity and local coordinate $\hat{x}$. Addition methods on the `Model` describe boundary conditions and a forcing term.

These `Model` classes are used for the implementation of both bulk ($\omega = \Omega$) and for surface ($\omega = \Gamma$) finite element methods. The lower diagonal block D representing the main part of the boundary pde is of the same form as A given above and the coupling occurs through the forcing term:

$$\int_\Gamma D(x, v(x, \nabla_\Gamma v(x)) \cdot \nabla_\Gamma \varphi(x) + m(x, v(x), \nabla_\Gamma v(x))\varphi(x) = \int_\Gamma \left(\beta_0 u + \beta_1 \nabla u \cdot n\right)$$

With suitable function β_0, β_1 defined on the coupling surface. Therefore,

$$< Cu, \varphi > := \int_\Gamma \left(\beta_0 u + \beta_1 \nabla u \cdot n\right)\varphi \tag{21}$$

However, user defined parameters $\alpha_\Omega, \alpha_\Gamma, \beta_0, \beta_1$ are intended for future releases, and all the examples presented below will effectively have them set to one where they are present.

For describing problems to be discetized by the boundary element method a similar model class needs to be implemented. We provide a model for a far field Laplace and Helmholtz problem. For example, to solve a complex valued Helmholtz equation of the form

$$\begin{aligned} \Delta u + \omega u &= u_{\text{inc}} && \text{in } R^3 \setminus \Omega \\ u &= g && \text{on } \partial\Omega \end{aligned}$$

the model class has the following form:

© by the authors, 2017

C++ code

```
// FS: The complex function space, GP: the grid part used
template <class FS, class GP>
struct HelmholtzModel
{
  HelmholtzModel(double omega) ;
  struct IncidentData {
    void evaluate( const DomainType& x, RangeType& res ) const;
  };
  struct DirichletData {
    void evaluate( const DomainType& x, RangeType& res ) const;
  };
};
```

The two sub-structures are used to implement the Dirichlet data g and the incident wave u_{inc} in global coordinates.

A model class together with a grid is used to initilize a `Scheme` class which sets up the required types for the discrete spaces and solvers and in addition also holds the solution. The `Scheme` classes provide two methods `prepare` and `solve`, which sets up the right hand side vector and assembles and solves the system, respectively. In this module two such scheme classes are implemented: `FemScheme` for solving non-linear fem problems using the classes provided in the DUNE-FEM module, and the `BemScheme` providing the bindings to `Bem++`.

C++ code

```
template < class Model, SolverType solver=istl >
struct FemScheme {
  FemScheme(const typename Model::GridPartType &gridpart, const Model& model);
  void prepare();
  void solve();
};
```

In addition the scheme class provides methods for accessing the discrete function spaces and the discrete function storing the computed solution. The `SolverType` template argument can be used to specify the solver backend to be used. While in DUNE-FEM the bindings for a number of solver packages are available (DUNE-ISTL, `PETSc`, `UMFPACK`, `Eigen`) we use DUNE-ISTL for all our finite element computations. The `BemScheme` class has a very similar structure but without the option of choosing a solver backend since we use the H-matrix implementations available in the `Bem++` package.

Both our coupled problems given by (1) and (2) consist each of three terms: $A\,u, f$ and $B\,v$ and $D\,v, g$, and $C\,u$, respectively. The first two terms of each problem are covered by the `prepare` and `solve` methods on the scheme classes. To describe the full problem the coupling terms $B\,v, C\,u$ need to be implemented. To this end, the scheme classes are extended by one additional method:

C++ code

```
template <class OtherDiscreteFunctionType>
void couple(const OtherDiscreteFunctionType &otherSolution);
```

where the template parameter is the solution of the other problem. As already mentioned above, coupling is achieved using the DUNE-GRID-GLUE module. In this version of the module only a linear coupling between the two schemes is considered. This is to be extended in future releases.

The actual coupling mechanism is itself implemented in a `Scheme` class, taking the bulk and the surface scheme classes and then itself providing a `prepare` and `solve` method.

As described above this module provides three such implementations:
`GaussSeidelCouplingScheme` implementing equation (12), `JacobiCouplingScheme` for (11) and finally `GMResCouplingScheme` for equation (16), which may all be found in the folder

 © by the authors, 2017

dune-fem-bem-toolbox/dune/fem_bem_toolbox/coupling_schemes

Of course such coupling necessitates a number of different grid and function space types associated with the communication of data, and the setting-up of the glue objects, especially with the moving grids of the third case study described below. For convenience, all these types are grouped together with the original type declarations of the schemes themselves within a `traits.hh` file inside the source code folder `source` for each example application.

5 Parallel Coupling Approach

Underpinning all of the coupling approaches is the use of the **dune-grid-glue** module to create "glue" objects between the bulk and surface meshes allowing communication of solution values held on each.

For parallel runs, the MPI based solvers in DUNE exploit a (possibly user defined) partitioning of full volume or surface grids into different pieces, each of which is attributed to a different process, such that the matrix assembly and solution is performed independently on each process.

Now DUNE-GRID-GLUE will create distinct "glue objects" between each of these individual pieces. With one piece of the volume grid, and one piece of the surface grid on a particular rank, one glue object will be created (on the same rank) that essentially creates and holds a surface grid corresponding to the intersection of the bulk and surface grids it is gluing together – and which thus normally coincides with the surface mesh itself.

The glue object then provides an "iterator" that iterates over the "glue elements" of this glue intersection grid which each have pointers to the bulk and surface elements of the bulk and surface grids it joins.

All that is thus required to exchange information from the bulk problem to the surface one (or vice-versa), is thus to iterate through these glue elements, for each one evaluating (at the Lagrange or quadrature points) the bulk solution that needs to be communicated within the glue element (coincident with a face of its own bulk element), and using the glue element to then assign this information to the surface solution at the corresponding Lagrange point within the corresponding surface element or assemble the desired integral terms. In this version of the module the coupling matrices B, C are not assembled but the coupling is computed on the fly. For the current problems where inverting A, D is the dominant cost the required iterations over the partitioned surface grid is an acceptable overhead.

The Bem++ library, in both serial *and* parallel execution modes, always works with full, unpartitioned (surface) grids. The parallelization being achieved through use of the multi-threaded assembly and solver routines both with or without the use of H-matrices to further speed-up assembly Smigaj et al. [2015]. Clearly the first challenge is thus to allow the communication of the solution variables held on the "full" surface grid (for the `Bem++` operations), to and from the grid parts held on each process performing the DUNE calculations. As storage backend for the degrees of freedom ("dofs") the vector classes from the `Eigen` package already used extensively by `Bem++` are used.

These `Eigen` vector structures give the option of being setup using a chunk of specified raw memory so that special shared memory blocks accessed via a special "shared" dof vector class `EigenVector` may be used for reading and writing through the vector interface of `Eigen` and thus enabling a seamless exchange of data between the DUNE-FEM and the `Bem++` solvers and between different processes, but to `Bem++` they look just like a regular vector of unknowns over the whole of the surface grid.

This wrapper class is called `Dune::Fem::EigenVector` and implements the vector interface used in DUNE-FEM to store the degrees of freedom for discrete functions to be stored both over a full grid and a partitioned grid. This class combines the vector classes of `Eigen` together with a block of raw memory to store the vector elements.

The module Dune-Fem is flexible with respect to the storage structure of the degrees of freedom. So through simple adapter classes, Dune-Fem makes it possible to use a wide range of solver backends for storing dof vectors for discrete functions and it is straight-forward to use the vectors of `Eigen`. Thus since `Eigen` can make use of the shared memory blocks, accessing those through the Dune-Fem interface is seamlessly possible.

Of course, to `Bem++` these dof vectors look like a regular vector of unknowns over the whole of the surface grid it has been given previously, but because they are built on shared memory, each of the Dune MPI ranks may access parts of the vector to fill in/evaluate each on the partitioned mesh.

Furthermore, because the `Eigen` vector classes provide an option for the user to provide raw memory to use as storage space, by allowing the first rank of a parallel execution to arrive at the particular point to create a special piece of shared memory, whose address it then communicates to all the other ranks, all ranks may then pass this address to the constructor of a new eigenvector storage array which they each create, but the addresses of whose elements will then be the same for all ranks.

By letting all ranks read from, and "non-clashing" (see below) ranks write to these shared elements, communication of values is possible.

This is the principle means for collecting and distributing data required for the `Bem++` routines, which are all run from a single process (which creates multiple threads). Typically the Dirichlet data required as the right hand side of the bem problem would be accumulated into a shared vector by the various ranks, the Neumann data returned by the bem solve (initiated on the rank zero process) would similarly be put into a shared vector, from which each rank could then extract the values relevant to it.

The construction of the shared memory block is done in the class `SharedMemory`. Rank zero tries to construct a block with a randomly generated name using `boost::interprocess::mapped_region`. If this is successful, the name is communicated to all other processes and they all use the same block of memory to store the degrees of freedom. The class has the following constructor

C++ code

```
1  EigenVector ( unsigned int size = 0, const bool shared = false );
```

where the last argument can be used to determine if a block of shared memory is to be used. The following code snippet shows how to setup both a distributed and a shared discrete function for a given `DiscreteFunctionSpaceType`, e.g., a Lagrange space:

C++ code

```
1  typedef Dune::Fem::EigenVector<double> DofVectorType;
2  typedef Dune::Fem::VectorDiscreteFunction< DiscreteFunctionSpaceType, DofVectorType
       > > DiscreteFunctionType;
3  DiscreteFunctionSpaceType distributedSpace( distributedGridPart );
4  DiscreteFunctionSpaceType fullSpace( fullGridPart );
5  DiscreteFunctionType rankLocalDiscreteFunction("rank_local", distributedSpace );
6  DiscreteFunctionType sharedDiscreteFunction("shared", true, fullSpace );
```

Of course, appropriate synchronization of the filling/extracting data tasks is needed. While reading from a particular piece of memory does not usually pose any problems with multiple processes trying to access the same memory address, writing to such an address does. The synchronization of these tasks is performed by allowing each rank to run over its own specially created glue object between its particular gridpart, and the full grid (on which the bem operations are performed), putting-down or picking-up values at the positions given by the glue object in accordance with a special "dofrank" array which grants (or not) the permission to write to any particular memory address.

 © by the authors, 2017

Each rank builds up its own "dofrank" array, defined over the full grid, where the integer values at each dof (corresponding to a particular grid point of the full grid) are just the rank number of the highest rank whose individual gridpart has a coincident grid point there.

Thus in summary to minimize communication and synchronization costs the degrees of freedom of g_Γ are stored in a block of shared memory using `Boost`'s interprocess package. Note that g_Γ is a continuous Lagrange function so that some of the degrees of freedom are shared between processes. To avoid race conditions, a degree of freedom in the shared memory block is only written to by one process (the one with the lowest rank). The solution of the `Bem++` solver is also directly written to a block of shared memory. After the Neumann data (piecewise constant) is computed, the second glue object is used again to move the data from the full surface grid onto the distributed surface grid. Note that each step described above is almost completely parallel. The assembly routines and the solvers from the `Bem++` library are only called from one process which then sets up its own multiple threads to carry out the computations in parallel.

A particular element (corresponding to a particular dof) in a shared memory eigenvector may then only be modified by a process whose rank is equal to that held in the dofrank array at the same point. It is thus just the highest ranking process at any particular gridpoint who is allowed to fill in dof values at the corresponding point.

For the finite-element problems we use a distributed grid parallelization strategy based on MPI. The required communication for both the assembly and the solvers are directly available through the Dune-Fem module and no changes are required to the `FemScheme` class to use it for parallel runs. Although the Dune-Grid-Glue will work for an arbitrary partitioning of both the surface and the bulk grid, in this version we make the following assumption: for each process p, the part Ω_p of the bulk grid on processor p and the part Γ_p of the surface grid satisfy $\Gamma_p = \partial\Omega_p$. As a consequence of this restriction on the partitioning, surface and bulk entities that are glued together always reside on the same process so that no additional communication is required.

This module provides two load balancing handlers for the `partition` method of Dune-ALUGrid to enforce this type of matching partitioning. The first `SimpleLoadBalanceHandle` class uses a simple coordinate bisection approach to partition both the bulk and the surface grid. While this approach leads to a reasonably good partition of the surface the bulk partitioning suffers from bad connectivity at the origin. To improve the bulk partitioning this module also provides `ZoltanLoadBalanceHandle`. The partitioning generated by this class is identical to the simple partitioning described previously but combines this with `Zoltan`'s graph partitioner Boman et al. [2012] around the origin to reduce the number of shared vertices there. In summary, coupled surface and bulk finite-element simulations can be in parallel with little change to the serial code.

As befits the nature of boundary element formulations the `Bem++` library, however, provides only shared (and not distributed) memory parallelization and to assemble the boundary integral operators requires an unpartitioned mesh for parallel assembly. For the matrix inversion, `Bem++` uses the OpenMP multi-threaded solvers of the `Eigen` package and with the additional option of using internal H-matrix based solvers. Again the dof vectors for the right hand side and the solution of the boundary element problem need to be accessible from all threads. Combining the distributed grid approach used in the Dune packages with the solvers in `Bem++` requiring the full unpartitioned grid is challenging. To achieve this an additional scheme (`SharedBemScheme`) is available in this module. On each process this scheme is constructed using both the full surface grid Γ and a distributed grid $\Gamma = \bigcup_p \Gamma_p$. Discrete functions g_Γ, λ_Γ defined on the full grid are setup to be used with the solvers in the `Bem++` library. To setup the right hand side for the boundary element problem on the full grid the term $\Pi_\Gamma u$ in the `couple` method needs to be computed. Note that u will be distributed and so a process p will only "see" the part of u computed on the partition Ω_p.

First we use the glue objects setup between the partitioned bulk grid and the partitioned surface grid (used also in the fem-fem coupling example below) to transfer the bulk data onto Γ_p.

© by the authors, 2017

A second glue object defined between the part of the surface Γ_p onto the full surface Γ is now used to transfer the Dirichlet data on Γ_p to the discrete function g_Γ defined on Γ.

This two-stage process allows any surface problems requiring only finite elements to have the performance benefits of distributed grid parallelization, while allowing any BEM (surface) problems to work on undistributed grids.

6 Case Studies

Presented here are three example applications to illustrate the use of the new fem-bem toolbox, the source code for which may be found in the folders within

```
dune-fem-bem-toolbox/case_studies
```

as will be indicated. All experiments were carried out on an Intel Xeon(R) E5-2650v2, 2.6GHz CPU node with 8 physical cores. For completeness, and to further later discussions into parallel performance of the coupling methods, which will make reference to "efficiencies" as measured by the percentage "efficiency"

$$E = 100 \times \frac{\text{time to solve on one core}}{\text{core count} \times \text{solve time}} \tag{22}$$

two additional folders are included within the case studies as example FEM only and BEM only applications.

The first solves a simple real valued Dirichlet bounded Poisson problem inside a unit cube, whilst the second solves for the Helmholtz equation, (which will be introduced in the second case study) outside a unit sphere.

These two reference problems are briefly discussed here, but only in terms of their parallel performance efficiencies shown in Figure 1, Figure 2 and Figure 5(left), respectively. The source code for them may be found within folders

```
dune-fem-bem-toolbox/case_studies/fem_test
```

and

```
dune-fem-bem-toolbox/case_studies/bem_test
```

respectively, with the same general structure, build and running instructions as the three studies which will be investigated in detail below.

The BEM only test results were performed on the surface mesh `sphere-h-0.05.msh` of 11459 elements found in the `bem_test` data folder, while the FEM only test results used the `unitcube-3d.dgf` in the `fem_test` data folder with chosen element diameters of 1/52 the (unit) edge length.

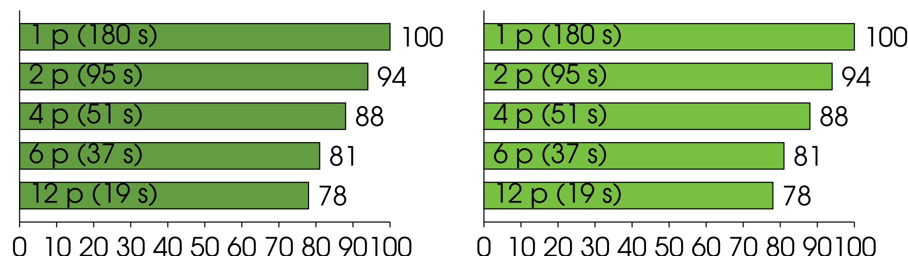


Figure 1: Percentage efficiencies, E, see equation (22) of operator assembly (left) and assembly+solve (right) times in seconds, "s", for solution of a BEM only Helmholtz problem using just `Bem++` *without* H-matrices for different processor/thread counts, "p".

 © by the authors, 2017

First we note, however, that the assembly of the operators within `Bem++` may be performed with or without the use of so-called "H-matrices". If they are not used, then an LU decomposition of the system matrix may be performed in the assembly step, leaving a simple back-substitution for the solve to do. This allows assembly and solve times as indicated in Figure 1 to be obtained with the test problem. Thus the solve step would appear to be instantaneous relative to the assemble step.

If, however, H-matrices are turned on, then a direct LU decomposition is not possible, and the solve step will involve a Krylov iterative solver and necessarily more work, and thus we obtain the solve times (and efficiencies) as appearing in Figure 2. Clearly the assembly times are much reduced with the H-matrices, but this is offset slightly by the more expensive solves.

This leads us to the conclusion that if repeated solution using the same basic BEM operators is going to be required, for instance in the repeated evaluations of a solution within a fixed-point iterative coupling scheme (as will be seen in the second case study), then it is perhaps best not to use the H-matrices in `Bem++`, because one LU decomposition, performed just once within the assembly stage, will allow very short times for many repeated solve steps within the iterative coupling scheme. In future, alternative more efficient parallelizations of the H-matrix assembly and solve steps Kriemann [2013], not yet implemented in `Bem++`, might change this advice however.

Now, if the BEM operators instead need to be reassembled often, as will be seen in the third case study, with no more solve steps than assembly steps, then H-matrices might be advantageous because of their speed up of the assembly process leading to generally significantly smaller overall combined assemble and solve times, compare Figure 1(right) with Figure 2(right).

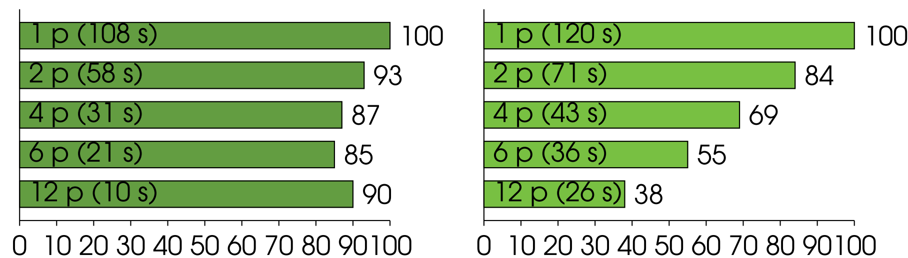


Figure 2: Percentage efficiencies, E, see equation (22) of operator assembly (left) and assembly+solve (right) times in seconds, "s", for solution of a BEM only Helmholtz problem using just `Bem++` *with* H-matrices for different processor/thread counts, "p".

It should also be noted that the `Bem++` H-matrix assembly of the BEM operators is very efficient with increasing parallelization, and near optimal, as can be seen in Figure 2(left). However, as soon as the solve stage to numerically invert these operators is taken into these timings as well, the parallel performance drops off significantly, see Figure 2(right).

The first case study will show a simple FEM-FEM iterative coupling on a toy problem, using the Gauss-Seidel approach, to illustrate performance with core count of a purely finite element problem for comparison.

The second and third examples will demonstrate real world problems with, respectively, the iterative GMRes scheme for the direct coupling of FEM and BEM in an electromagnetic scattering example, and an indirect FEM-BEM coupling for a fluid example, which may not be so easily cast into the form of equations (12), (11) or (16), but illustrates the future possibilities for many other types of FEM-BEM coupling.

For additional clarity, the key governing equations and boundary conditions in the following have been labelled with "FEM" or "BEM" to indicate the discretization scheme used for the former, and

© by the authors, 2017

"Dir", "Neu", "Rob" and "Rad" for the Dirichlet, Neumann, Robin and Radiation type boundary conditions of the latter.

Instructions on running the examples are included after each problem description, but of course, this should not be attempted until after a successfull installation of the software, for which the reader is referred to appendix B.

6.1 Bulk-Surface Fem-Fem Coupling Example

The source code for the first case study may be found within the following folder:

```
dune-fem-bem-toolbox/case_studies/coupled_sphere
```

Notes on how to run this example will be presented after the following problem description.

Here a volume (or "bulk") solution, $u \in \Omega$, is coupled to a surface solution on Ω's surface, $v \in \Gamma \equiv \partial\Omega$. Both solutions being represented by piecewise linear and globally continuous finite element basis functions.

The coupling, through the boundary conditions, and governing equations are Ranner and Elliott [2013]:

$$\begin{aligned} FEM: &\quad -\Delta u + u &&= f &&\quad in\ \Omega && (23a)\\ Rob: &\quad u - v + \frac{\partial u}{\partial n} &&= 0 &&\quad on\ \Gamma && (23b)\\ FEM: &\quad -\Delta_\Gamma v + v + \frac{\partial u}{\partial n} &&= g &&\quad on\ \Gamma && (23c) \end{aligned}$$

The forcings f and g must, of course, be compatible, and may be found from any choice of analytic solutions for u and v. In the following, we chose the exponential solutions:

$$\begin{aligned} u(x,y,z) &= exp\left[-x\,(x-1) - y\,(y-1)\right]\\ v(x,y,z) &= \left[1 + x\,(1-2\,x) + y\,(1-2\,y)\right]\\ &\quad \times exp\left[-x\,(x-1) - y\,(y-1)\right] \end{aligned} \quad (24)$$

which may easily be seen to satisfy the governing equations and boundary conditions.

The weak formulation of (23c) suggests an iterative coupling of the form

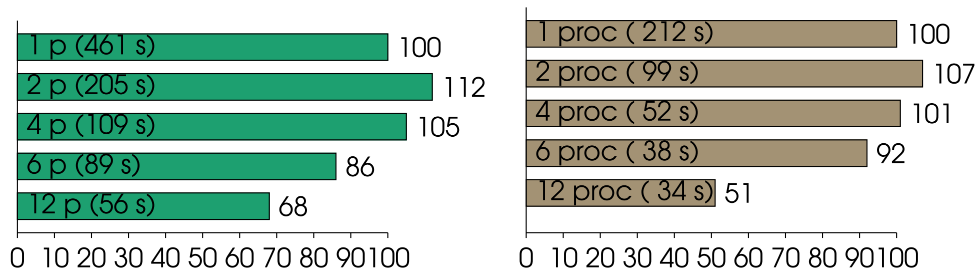


Figure 3: Percentage efficiencies, E, see equation (22) and solve times in seconds, "s", for solution of the coupled sphere example with element diameter ("cellsize") 0.019, giving 1924626 elements, see § 6.1, using just DUNE modules (left) and for solution of the charged droplet example (right) on the "maxi" meshes of 18239 elements, see § 6.3 later, using both DUNE and Bem++, for different processor/thread counts, "p".

 © by the authors, 2017

$$\underbrace{\int_\Omega (\nabla u \cdot \nabla \eta + u\eta) + \int_\Gamma u\eta}_{Au} = \underbrace{\int_\Omega f\eta}_{f} - \underbrace{\int_\Gamma -v\eta}_{Bv}$$
$$\underbrace{\int_\Gamma (\nabla_\Gamma v \cdot \nabla_\Gamma \xi + v\,\xi) + \int_\Gamma v\xi}_{Dv} = \underbrace{\int_\Gamma g\xi}_{g} - \underbrace{\int_\Gamma -u\xi}_{Cu}$$

which clearly shows the matrix form of equation (3) where $\eta \in V_\Omega$ and $\zeta \in V_\Gamma$ are first order Lagrange finite element in the bulk and on the surface, respectively. We choose to use the Gauss-Seidel coupling scheme of (12), but the user could equally well try one of the other coupling schemes.

Table 1: Convergence rates for bulk-surface example.

Refinement Level	Iteration Count	Error	Time (secs)	Convergence Order
0	20	4.09689	2	-
1	19	0.702664	2	2.54362
2	18	0.229774	6	1.61262
3	18	0.0729218	32	1.65579
4	18	0.0205689	229	1.82588
5	18	0.00548633	1952	1.90656
6	18	0.00144776	19900	1.92201

Because of the availability of an analytic solution to this problem, shown in equations (24), it is possible to calculate standard $\mathbb{L}^2$ error norms for the solution, which can then be tested for the optimal second order convergence with grid refinement expected of linear solution interpolations.

In Table 1 can be seen the convergence orders of a series of six such grid refinements, together with indications of the iteration counts and solve times it took to achieve them on a single processor.

Clearly the convergence orders are about where one would hope to see them for a single FEM problem using linear elements, indicating that nothing has been “lost” in the suggested coupling scheme adopted for two such problems.

Having now shown that the coupling of two schemes had no adverse impact on the error convergence rates with mesh refinement, we now check that coupling has no serious impact on parallel (multi-core) performance either, as measured by the efficiency defined in equation (22).

In Figure 3 (left) can be seen the percentage efficiencies, E, and solve times in seconds, for the solution on a single Intel Xeon node using 1, 2, 4, 6 or 12 cores of the bulk-surface coupled sphere problem with a chosen finite element diameter, or “cellSize” (see below) of 0.019.

Running the example: Within the example’s `source` folder type `make main` at the command prompt. This will build the executable. To run the code in serial type `./main`; for parallel use type `mpiexec -np N main` where “N” is the number of processors/threads you would like to use. The volume mesh used for the example is specified within the `parameter` file within the `data` folder on the line beginning with `fem.io.macroGridFile_3d:`.

The surface mesh is generated automatically from the volume mesh when running `main`, however, due to issues within the grid management software on which the present module depends, this automatic surface generation is not possible in parallel. For parallel operation the user is recommended to run first in serial with surface.use:false on the command line (overiding that in the parameter file), thus `./main surface.use:false`, to generate the surface mesh, and then

© by the authors, 2017

ensure `surface.use: true` within the parameter file. This will then use the previously created surface grid with user specified location set by `fem.io.macroGrydFile_2d: surface.dgf`.

For convenience, the software is distributed with relatively small volume and surface meshes pre-prepared. For higher grid resolutions the user should adapt the "cellSize" within the `sphere.gmsh` file in the `data` folder, and then use the `gmsh` software Geuzaine and Remacle [2009] installed into the misc directory to generate a mesh file "sphere.msh", using the command

```
../../../misc/gmsh-2.8.4-Linux/bin/gmsh sphere.gmsh -3 -v 0 -format msh -o sphere.msh
```

typed from within the `data` folder.

To view the grid based "vtu" output that will have been placed inside the `output` folder after program execution, the `paraview` software Ayachit and Utkarsh [2015] is required to be installed.

Finally, the efficiency tests may be run by typing `./eff.bot` at the command prompt. For these tests the user need only adapt the "cellSize" mentioned above for checking the efficiencies on different meshes, the mesh file generation and surface grid creation are done automatically by the script.

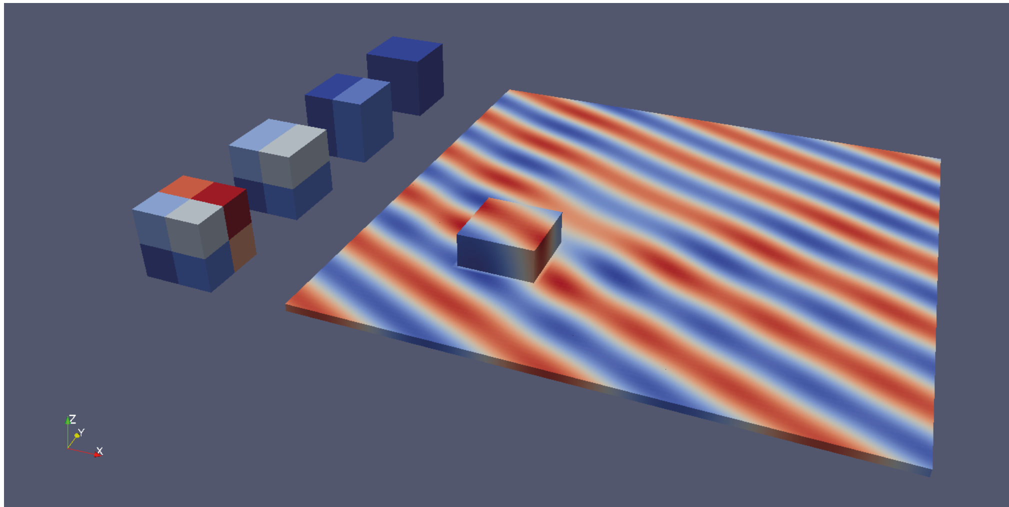


Figure 4: Real component of electromagnetic wave scattering solution to equation (25) with $k = 12$ on the surface of the box corresponding to Ω (see text) with finite element diameters 1/16 edge length, and projected onto a surrounding plane by the exterior representation formulae together with multi-core grid partitioning examples.

6.2 Direct Coupling: Scattering Electro-Magnetic Wave Example

The source code for the second case study may be found within the following folder:

dune-fem-bem-toolbox/case_studies/scattering_wave

Notes on how to run this example will be presented after the following problem description.

The classic Fem-Bem coupling example of an incident wave hitting a target, with a Fem scheme defined for the Helmholtz equation of the "total" solution – equal to the "incident" plus "scattered" solutions – with possibly spacially varying coefficients, over the finite "interior" domain of the target – a cube $\Omega = [-0.5, 0.5]^3$ in our case, and corresponding to the central box in Figure 4 – coupled to a Bem scheme on $\partial\Omega$ supporting a scattered solution to the same equation, but with constant coefficients, in the "exterior" where the incident wave originates.

 © by the authors, 2017

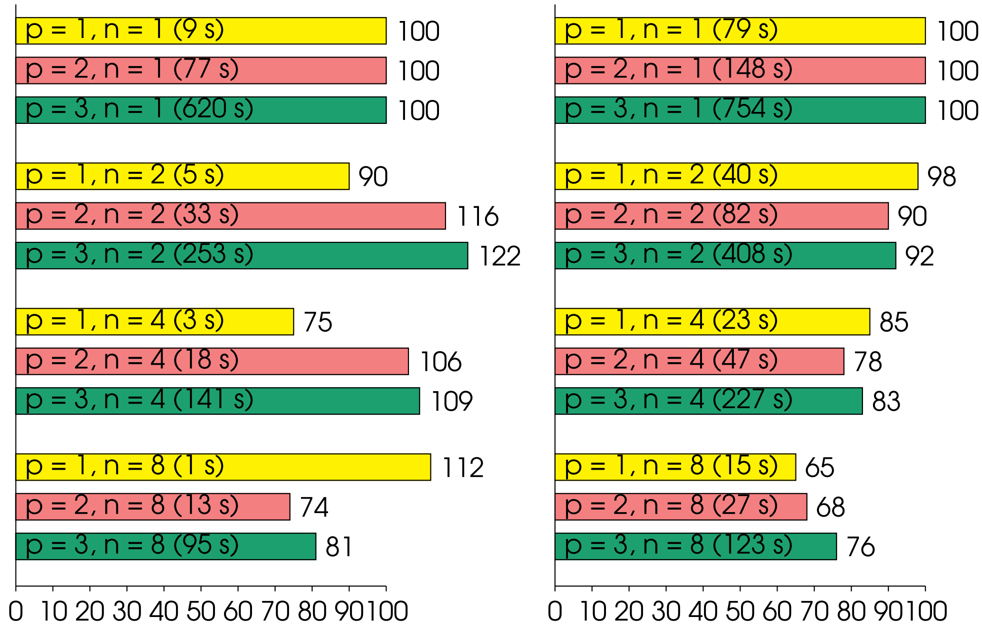


Figure 5: Percentage efficiencies, E, see equation (22), (and solve times in seconds, "s") for the solution of a simple Poisson example system (left) and system (25) in Ω with finite element diameters 1/8 edge length and $k = 6$ (right) for different core counts, "n", and interpolation polynomial orders, "p".

Now while the Bem solution is always represented by piecewise linear globally continuous basis functions (those supported by `Bem++` at present), because of the system of coupling adopted, the interior solution may use linear, quadratic and even cubic basis functions.

The values of this exterior (total) solution on an arbitrary plane around the box (discretized with linear finite elements of diameter 1/16 the side length) are also shown in Figure 4, and were computed via the appropriate exterior representation formula for the scattered field (which will be seen to be the starting point for the BEM formulations next) plus the incident field.

We require that the total solution, and its normal derivative, are continuous across the interior/exterior boundary $\partial\Omega$, and that the total (exterior) solution tends to a plane wave towards infinity at a rate inversely proportional to the distance, giving the governing equations and boundary conditions as

$$\begin{aligned}
FEM: &\quad \Delta u + n(\mathbf{x})\,k^2 u &&= 0 && in\ \Omega && (25a)\\
BEM: &\quad \Delta v + k^2 v &&= 0 && in\ \mathbb{R}^3\backslash\Omega && (25b)\\
Dir: &\quad u - v &&= 0 && in\ \Gamma && (25c)\\
Neu: &\quad \frac{\partial u}{\partial n} - \frac{\partial v}{\partial n} &&= 0 && in\ \Gamma && (25d)\\
Rad: &\quad \left|\frac{\partial v}{\partial r} - i\,k\,v(x)\right| &&= O\left(|x|^{-1}\right) && as\ \ r \equiv |x| \to \infty && (25e)
\end{aligned}$$

where u and v are the interior and exterior solutions, k the exterior wave-number, and $n(\mathbf{x})$ provides a spacially varying coefficient for the interior

© by the authors, 2017

$$n(\mathbf{x}) = \frac{1 - 0.5\exp\left(-\|\mathbf{x}\|_\infty^2\right)}{1 - 0.5\exp(-0.25)} \tag{26}$$

Clearly this coefficient is 1 on the boundary and decreases towards the centre of the cube, representing an object which is denser towards its centre.

Let the incident wave be given by

$$u^{\text{inc}} = e^{ik\mathbf{x}\cdot\mathbf{d}} \tag{27}$$

where **d** gives the direction of wave travel.

In the exterior, v can be written as $v = v^{\text{inc}} + v^{\text{sca}}$ where $v^{\text{inc}} \equiv u^{\text{inc}}$ is the incident wave and v^{sca} is the scattered wave it generates from the target.

Following the classic boundary element derivations, see Colton and Kress [1983], for the discretization of equation (25b) in the exterior region $\Omega^\infty = \mathbb{R}^3\backslash\Omega$, we start with Green's Exterior Representation Formula – which is only valid for scattered fields and is used for the plotting on the plane mentioned above – for this exterior region

$$u^{sca}(x) = -\hat{V}\psi + \hat{K}\phi \quad \forall x \in \Omega^\infty \tag{28}$$

in terms of single, $\hat{V}$, and double, $\hat{K}$, layer potential operators defined in the appropriate Hilbert spaces

$$\hat{V} : H^{-1/2}(\Gamma) \to H^1(\Omega^\infty) \qquad \hat{K} : H^{1/2}(\Gamma) \to H^1(\Omega^\infty)$$

and acting on total Dirichlet, ϕ, and Neumann, ψ, data located on the surface of the exterior region Γ.

Now "bringing down" the unknown, $u^{sca} = u^{sca}(x)$, on to the surface, Γ, and then adding the incident field in order to identify it with the total Dirichlet data there, introduces a jump to the double layer operator, and equation (28) becomes

$$\underbrace{V\psi}_{Dv} = \underbrace{u^{\text{inc}}}_{g} - \underbrace{\left(\frac{1}{2}I - K\right)\phi}_{Cu} \quad \forall x \in \Gamma \tag{29}$$

where the operators have taken on the true BEM forms

$$V : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma) \qquad K : H^{1/2}(\Gamma) \to H^{1/2}(\Gamma)$$

Note that ϕ is the Dirichlet data defined on the boundary grid which due to the coupling conditions is the trace of the solution u from the bulk solution. Since `Bem++` only accepts piecewise linear Dirichlet functions computing $C\,u$ involves the projection of the trace of the bulk solution onto the space of piecewise linear functions on the surface grid. Having established the BEM formulation, the FEM derivations start with the weak form of equation (25a)

$$\underbrace{\int_\Omega \nabla u \cdot \nabla\eta - k^2 \int_\Omega n^2 u\,\eta}_{A\,u} = \underbrace{0}_{f} + \underbrace{\int_{\partial\Omega} \psi\eta}_{B\,v} \tag{30}$$

where $\psi = \frac{\partial u}{\partial n}$ is identical to the Neumann data of the BEM formulation.

With the corresponding matrices A, B, C and D of equation (3) for this problem as indicated, where the Dirichlet data ϕ is identical to the solution u of the first problem on the boundary Γ, and the

 © by the authors, 2017

Neumann data ψ is chosen as the solution v of the second problem, the coupling scheme (16) of section 2 with a GMRes Krylov iterative solver may then be used to find a solution to the complete problem.

For parallel processing in the numerical implementation of the scattering problem, the cube mesh is subdivided into block shaped pieces, as demonstrated to the left of Figure 4, with assembly and solution of the system matrix corresponding to each piece performed by the FEM scheme with one piece per rank using MPI communications to keep the calculations in sync.

The mesh of the surface of the cube, however, is not subdivided, but passed to the bem scheme whole. The use of multiple threads within the BEM solver is performed via the TBB libraries.

Glue objects are created, one per rank, to transport Dirichlet and Neumann data between the whole surface grid used by the BEM scheme, to the portion of that surface coincident with the surface of the volume grid piece located on the particular thread.

As might be expected from the nature of the spacially varying coefficient for the interior seen in equation (26), the image seen in Figure 4 shows the diffraction of the incident wave as it passes through the cubic block of material with the spacially varying density, calculated after 74 iterations of the coupling scheme. This generates the classic "spokes" of increased and decreased wave amplitudes radiating away from the diffracting object itself. More discussion of such phenomena may be found in Heald and Marion [2012].

As with the previous example, an efficiency test is also provided for the present scattering wave example, the results of which are shown in Figure 5 (right) for the calculation on 1, 2, 4 or 8 cores of the full system (25) but now with finite element interpolation polynomials of orders 1 (yellow), 2 (pink) and 3 (green) for the interior.

To keep computation times practical for easy reproduction by the end-user when trying the cubic interpolations, the finite element diameters used in the efficiency tests here are double those used for Figure 4, with the wavenumber k halved to keep the solutions meaningful on this coarser discretization – it is generally recommended to have at least 5-6 (linear) elements per wavelength in numerical computations.

As indicated above, because for this example a fixed-point iterative coupling scheme is used, the H-matrix option of `Bem++` was turned off, because the solve steps need to be repeated many times during the convergence process, but they only require one assembly performed right at the beginning. However, it can be seen that the parallel performance is just slightly worse than with the previous example, where only FEM calculations were involved, or similarly with the FEM only test (simple real valued Poisson example on a unit cube), whose results, with element diameters of 1/52 edge length, are placed in Figure 5 (left) for direct comparison.

It is also worth noting the slight but significant improvement in efficiencies with the higher order interpolation polynomials. This is attributed to increases in the overall problem size, and thus lower communication costs between the cores, relative to their overall workloads.

Running the example: Within the example's `source` folder type `make main` at the command prompt. This will build the executable. Adapt the `parameter` file in the `data` folder to the settings required – the user may even turn the H-matrices on if curious as to their effect by setting `bempp.hmatrix:true` – the parameters prefixed with "helmholtz" will control the incident field to be used (via "helmholtz.omega" for the wave number and "helmholtz.amplitude" for the amplitude) and the characteristics of the box density via the "helmholtz.boxwave" parameter. If the latter is set to zero, then the result shown in Figure 4 for the density of equation (26) will be obtained, while alternative values n greater than zero will give the box a constant relative density of n relative to the surrounding medium. To run the code in serial type `./main`; for parallel execution use `mpiexec -np N main` where "N" is the number of threads you would like to use. Note that the `Bem++` part usually just uses however many processors it can find, which is fine for most situations, however, to specify the parallelization completely, the `Bem++` thread count environment variable should be set with `export BEMPP_NUM_THREADS=N`, as is done in the

`eff.bot` script for efficiency testing. The volume mesh used for the example is specified within the `parameter` file within the `data` folder on the line beginning with `fem.io.macroGridFile_3d:`. The surface mesh is generated automatically from the volume mesh, however, due to issues within the grid management software on which the present module depends, this automatic surface generation is not possible in parallel. For parallel operation the user is recommended to run first in serial to generate the surface mesh, with `surface.use: false` within the parameter file, or on the command line as `./main surface.use:false`, and then reset `surface.use: true` afterwards for all subsequent operations which will then use the previously created surface grid with user specified location set by `fem.io.macroGrydFile_2d: surface.dgf`.

To alter the refinement of the mesh, the grid file `unitcube-3d.dgf` in the `data` folder may be adapted, and a new surface mesh will then be required by following the above mentioned steps. For convenience the code is distributed with the surface mesh matching the 16x16x16 subdivision in `unitcube-3d.dgf`, which is that used for Figure 4. The grid refinement chosen should bear in mind the wavenumber, k, of solution to be calculated – high wavenumbers computed on overly coarse grids can lead to unexpected results, or even non-convergence of the solver.

The type of grid based "vtu" output may be chosen by the user through the `external_grid.level` parameter; -10 will give no output, -1 will return the vtu output just for the solution within the box, while 0 will give both the solution within the box domain and the exterior projection onto the surface plane as seen in Figure 4. Note that vtu output will have been placed inside the `output` folder, and the `paraview` software is required to be installed in order to view it.

Finally, the efficiency tests may be run by typing `./eff.bot` at the command prompt.

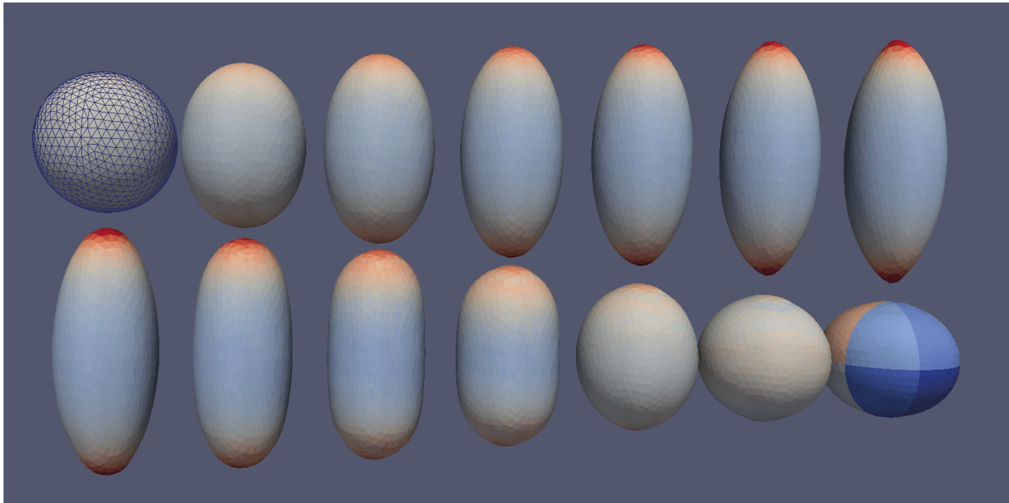


Figure 6: Non-dimensional surface charge concentration on the deformed droplet at various times of extension (top row) and relaxation (bottom row). The mesh and grid partitioning, into 12, used for the actual calculations are as indicated in the first and last images respectively. Note the point of maximum deformation – and the formation of a conical tip Radcliffe [2013] – in the top right image.

6.3 Indirect Coupling: Charged Droplet Example

The source code for the third, and final, case study may be found within the following folder:

dune-fem-bem-toolbox/case_studies/charged_droplet

Notes on how to run this example will be presented after the following problem description.

 © by the authors, 2017

This is a more advanced example application and shows use of the toolbox outside of the direct FEM-BEM coupling methods of (12), (11) or (16), with the FEM and BEM schemes now computing very different quantites and the coupling between the two being less direct or immediate than in the previous examples.

As with the previous case, however, a BEM solution at one iteration provides Neumann data directly for use as a boundary condition to the next iteration of the FEM scheme. However, the return influence of the FEM on the BEM, while still using Dirichlet data as previously, is slightly more sophisticated.

Instead of the Dirichlet data on the boundary from a FEM solve simply being given directly to the BEM scheme, in this example it is used instead to move the actual boundary Γ on which the BEM integral kernels themselves are calculated. The BEM calculation itself on this new boundary then just uses a constant value in place of the Dirichlet data it might otherwise have received directly from the FEM.

Because the boundary Γ is constantly moving, this iterative strategy does not converge to a fixed point as previously, but provides a simple yet effective means to solve a real world moving boundary problem of great interest.

Furthermore, this motion of the boundary on which the BEM integrals are calculated necessitates a reassembly of the BEM operators every timestep too. For this reason the H-matrix option of `Bem++` mentioned above is used to significantly reduce the operator assembly times.

The exact details of the problem may be found in Radcliffe [2013], but essentially a bulk, incompressible, Navier-Stokes volume FEM scheme is used to determine the vector fluid velocities, $\mathbf{u}$, of an initially spherical (with radius r) droplet interior, $\Omega \subset \mathbb{R}^3$, driven by the normal, $\mathbf{n}$, directed surface tractions on its boundary Γ arising from both a scalar electric surface charge distribution (proportional to the normal derivative of a Laplacian solution v, calculated by a surface BEM scheme) and from classical surface tension.

The strength of this surface tension at any time and position is found by means of a second FEM scheme, but this time just for the surface Γ, solving the classical "mean curvature flow" problem which gives the vector solution, $\mathbf{w}$, corresponding to the new positions each point of Γ would take if displaced in the local normal direction by a distance proportional to (twice) the local mean curvature.

Given that the required surface tension strength is directly proportional to local mean curvature, and acts in the normal direction, the mean curvature flow solution $\mathbf{w}$ can thus be used immediately in the calculations.

Indeed, it is remarked here that the orientation of the displacements in the FEM calculated vector $\mathbf{w}$ after each time increment is actually what is used to establish the normal direction $\mathbf{n}$ in a robust manner in the computations themselves. With a simple first order semi implicit time discretization of the Navier-Stokes equations – and in contrast to Radcliffe [2013], using linear piecewise continuous basis functions with a simple pressure stabilization – the problem may be cast into the form of the previous examples by a multiplication through by the timestep τ.

With initial conditions of $\mathbf{u} \equiv \mathbf{0}$ and $\mathbf{w} = \mathbf{x}(\Gamma)$, the initial (discretized) position of Γ, the governing equations for the scalar v, and the vector $\mathbf{u}$ & $\mathbf{w}$, valued unknowns (with a minus "-" superscript

indicating values at a previous timestep) are then

$$
\begin{aligned}
&FEM: && -\kappa\,\Delta\,\mathbf{u}+\rho\,\mathbf{u} && = L(\mathbf{u}^-) && in\,\Omega && \text{(31a)}\\
&FEM: && \nabla\cdot\mathbf{u} && = 0 && in\,\Omega && \text{(31b)}\\
&FEM: && -\tau\,\Delta_\Gamma\,\mathbf{w}+\mathbf{w} && = \mathbf{w}^- && on\,\Gamma && \text{(31c)}\\
&Rob: && \frac{\partial\mathbf{u}}{\partial n}-a\,\mathbf{w}+\mathbf{b}\left|\frac{\partial v}{\partial n}\right|^2 && = -a\,\mathbf{w}^- && on\,\Gamma && \text{(31d)}\\
&BEM: && \Delta\,v && = 0 && in\,\mathbb{R}^3\backslash\Omega && \text{(31e)}\\
&BEM: && v && = Const. = 1 && on\,\Gamma && \text{(31f)}
\end{aligned}
$$

where we have scalar, κ & a, and vector, $\mathbf{b}$, valued coefficients involving the density, ρ, surface tension, γ, dynamic viscosity, μ, total surface charge Q, and the electrical permitivity of free space, ϵ_0, respectively

$$\kappa=\tau\,\mu\,\theta \qquad a=\frac{\gamma}{\tau} \qquad \mathbf{b}=\frac{Q^2}{r^3\epsilon_0 q^2}\,\mathbf{n}$$

Here $q=\int_\Gamma \partial_n v\,ds$ just allows us to remove the influence of the otherwise arbitrary constant = 1 in (31f) and $\theta\in[0,1]$ determines how implicitly the Laplace term is to be treated – the associated source code has $\theta=0.5$ at present – and the linear operator L is given by

$$L(\mathbf{u})=\rho\,\mathbf{u}+(\tau\,\mu-\kappa)\,\Delta\,\mathbf{u}-\tau\,\rho\,(\mathbf{u}\cdot\nabla)\,\mathbf{u}-\tau\,\nabla\,p \tag{32}$$

with a pressure, p, arising as the Lagrange multiplier for the incompressibility condition, equation (31b), in the usual manner.

Non-dimensionalization on a unit radius drop, $r\equiv 1$, via the viscocity and charge by setting $\mu\equiv 1$, indicates that there are just two *independent* parameter groupings, $Oh=\mu/\sqrt{\rho\,r\,\gamma}$ and $\chi=Q^2/\left(64\pi^2\epsilon_0\gamma\,r^3\right)$ allowing $\rho\equiv\gamma\equiv Oh^{-1}$, see Radcliffe [2013], and thus

$$\kappa=\tau\,\theta \qquad a=\frac{1}{\tau\,Oh} \qquad \mathbf{b}=\frac{64\pi^2\chi}{Oh\,q^2}\,\mathbf{n}$$

It is interesting to note that here we only have one formal boundary condition, equation (31d), that involving the normal derivative of the (interior) solution u that allows the BEM to influence the FEM.

As indicated above, the return influence of the FEM on the BEM is still via the undifferentiated solution, however it is just that this solution (now the interior fluid velocity) instead advects the very boundary on which the BEM integrals are located.

Following the BEM discretization, see equation (29), of the Helmholtz equation (25b) for the previous example, the BEM form of the much simpler Laplace equation (31e), which has a solution in the form of a potential, is similar, but lacks the double layer operator or incident field and uses constant Dirichlet data $\phi=Const.$

$$\underbrace{V\psi}_{D\,v} = \underbrace{0}_{g} - \underbrace{(-I)\,\phi}_{C\,u} \quad \forall x\in\Gamma \tag{33}$$

Thus we simply invert the single layer potential operator, V, onto constant Dirichlet data, $\phi\equiv v$, to obtain the Neumann data, ψ, that is needed to determine the surface charge – all the variation

 © by the authors, 2017

in this Neumann data comes from the shape of the boundary Γ on which the V was calculated, rather than from the Dirichlet data it acts upon.

For the non-dimensionalization above to work, the constant *Const.* should ideally be such that the integral over Γ of v is equal to one. However, owing to the linear nature of equation (33), in practice it is preferable to simply use unit constant Dirichlet data, $\phi \equiv Const. = 1$, as in (31f), and just divide the resulting $v \equiv \psi$ distribution by whatever the integral over Γ of v turns out to be, or the q mentioned above.

Given the great similarities between equations (31a) and (31c) with equation (25a), it follows that their FEM forms are almost identical

$$\underbrace{\int_\Omega \kappa \nabla u \cdot \nabla \eta - \int_\Omega \rho\, u\, \eta}_{A\,u} = \underbrace{\int_\Omega L\,(\mathbf{u}^-)\,\eta}_{f} - \underbrace{\int_\Gamma \mathbf{a}\,(w - w^-)\,\eta + \int_\Gamma \mathbf{b}\, v^2 \eta}_{B\,v} \tag{34}$$

which is accompanied by the FEM form of the incompressibility condition (31b)

$$\int_\Omega \nabla \cdot \mathbf{u}\, p = 0 \tag{35}$$

in the volume, and on the surface there is

$$\int_\Gamma k \nabla w \cdot \nabla \eta - \int_\Gamma w\, \eta = w^- \tag{36}$$

for the FEM discretization of equation (31c).

The possible matrices A, B, C and D of equation (3) have been indicated for completeness but, as described above, the iterations in this example do not lead to a fixed point solution, but rather a sequence of solutions that may be seen in Figure 6, computed using a relatively small mesh of 1400 elements (see "Running the example" below).

Note that to follow the changes in geometry, the computational mesh used for the calculations was required to move between iterations. Now while the calculations for v and $\mathbf{w}$ are dependent only on the instantaneous shape of Γ at any particular iteration, the solution $\mathbf{u}$ involves a "history" bound-up in the operator L on the solution at the previous iteration $\mathbf{u}^-$. To accomodate the fact that this $\mathbf{u}^-$ is defined on a (slightly) different mesh, an ALE technique was adopted with a Laplacian mesh smoothing algorithm to set the interior mesh points once the Γ position had been updated, see Radcliffe [2016] for the details. Within the code this necessitated a further simple FEM scheme for the Laplacian operator to calculate these interior mesh node positions.

In Figure 6 can be seen a series of snapshots showing the evolution of the Γ boundary over time. In the top row of images electric charge, v, accumulates at the top and bottom of the droplet, causing it to deform from the sphere towards the rightmost image, where the parameter χ is reduced from 1 to 0.7 to correspond to an expulsion of charge in a liquid jet – images from experiments may be seen in Giglio et al. [2008] – which incurrs a little fluid loss Radcliffe [2016]. This reduction in overall charge loading allows surface tension, $\propto w$, to regain dominance and restore the droplet shape back to a sphere again in the lower set of images. More discussion on this very interesting behaviour may be found in Radcliffe [2013, 2016].
As with the previous examples, efficiency testing has also been performed here, with the results shown in Figure 3 (right), where they compare very favourably with those of the first case study which didn't involve `Bem++` at all, Figure 3 (left). Note that, to give the parallelization "more to work on" a more refined "maxi" mesh of 18239 elements was used, but, to keep the overall running time of the efficiency tests reasonable, a very short end time (0.01) was adopted.

© by the authors, 2017

Thus we may conclude that the parallelization of both the assembly (using H-matrix methods) of the BEM operators, and their solution via Krylov methods drawn from the `Eigen` library has gone quite well, with no significant impact on parallel performance.

Running the example: Within the example's `source` folder type `make main` at the command prompt. This will build the executable. To run the code in serial type `./main`; for parallel use `mpiexec -np N main` where "N" is the number of threads you would like to use. Note that the `Bem++` part usually just uses however many processors it can find, which is fine for most situations, however, to specify the parallelization completely, the `Bem++` thread count environment variable should be set with `export BEMPP_NUM_THREADS=N`, as is done in the `eff.bot` script for efficiency testing. Along with various parameters controlling the dynamics of the droplet and explained in comments above each one, the volume mesh used for the example is specified within the `parameter` file within the `data` folder on the line beginning with `fem.io.macroGridFile_3d:`. The user may create their own volume meshes with varying element diameter ("cellSize"), say, by adapting the `sphere.gmsh` mesh command file and then using the mesh generation software `gmsh` Geuzaine and Remacle [2009] installed into the misc directory to execute these commands and generate a mesh file "sphere.msh", using the command

```
../../../misc/gmsh-2.8.4-Linux/bin/gmsh sphere.gmsh -3 -v 0 -format msh -o sphere.msh
```

typed from within the `data` folder.

The surface mesh is generated automatically from the volume mesh when running the `main` program, however, due to issues within the grid management software on which the present module depends, this automatic surface generation is not possible in parallel. For parallel operation the user is recommended to run first in serial with `surface.use: false` within the parameter file, or on the command line as `./main surface.use:false`, to generate the surface mesh, and then return `surface.use: true` for the parallel runs. This will then use the previously created surface grid with user specified location set by `fem.io.macroGrydFile_2d: surface.dgf`.

To view the grid based "vtu" output that will have been placed inside the `output` folder, the `paraview` software is required to be installed. To adapt to different speed/memory/result requirements, different amounts of output are possible, specified by `vtu.output: n`, where "n" indicates the level required. n=0 gives no output; n=1 just the surface charge and n>10 all output.

For convenience the code is distributed with two sets of volume and surface meshes: a "mini" set involving 1400 elements for reproducing the results shown in Figure 6 in a moderate amount of time with a `droplet.timestep: 0.01`, and a more refined "maxi" set using 18239 elements that may be used for the efficiency tests which may be run by typing `./eff.bot` at the command prompt; with a recommended `droplet.timestep: 0.0023`. The latter command will run the simple bash script within the file `eff.bot`. For the efficiency testing it is suggested the user adapt the `droplet.endtime:` within the `parameter` file to something of their preference less than 3 or so – a value of just 0.01 was used for Figure 3 (right). At present values of 15 or above will allow reproduction of the full deformation pathway with the "mini" meshes as shown in Figure 6 – provided the appropriate vtu output is chosen – but might be a bit time consuming for the lower processor counts in efficiency testing, where it is additionally noted that such vtu output should be turned off for optimal performance.

7 Conclusions

A program structure has been presented that allows the combined use, within the same computation, of two computational discretization schemes, FEM and BEM, with very different computational architecture requirements for optimal performance.

Based on local interactions between the values of the discretized variables at the mesh points, the finite element method is well suited to distributed memory machines, while the global interactions between such discretized values of the boundary element method are best accomodated on shared memory architectures.

 © by the authors, 2017

By the asyncronous evaluation of one scheme, and then the other, with communication steps between, it is possible to have optimal environments for each scheme on a machine with just shared memory.

A computational toolbox for the development of such programs has been presented, with three-dimensional examples of the electromagnetic scattering off a box, and the deformation of a charged droplet used to give an indication of the performance improvements with processor thread count.

In this first version of the module some restrictions still apply most notably on the partitioning of the bulk and surface grid. This restriction will be removed in a further release of the module. The use of the DUNE-GRID-GLUE module also allows us to use non matching surface and bulk grids. We have not made use of this feature in the test cases presented here but we have first results for the wave scattering problem with finer bulk grids which look promising. Finally fully non-linear coupling is not yet possible and will be addressed in the next release of this module. This will also allow us to perform on larger scale distributed memory machines for the bulk problem. Since `Bem++` requires an undistributed grid, the surface would still be located on a single node and distributed between threads based on the approach described above. The bulk domain could be distributed differently and make use of multiple cores. This will require some additional investigation with respect to optimal load balancing of the coupled problem.

Finally this module still uses the 2.3 release of the DUNE modules. This is due to the fact that `Bem++` is still based on this release. As soon as `Bem++` has been moved to use the newer 2.4 DUNE release, then a 2.4 release of this module should follow shortly afterwards.

A Structure of this module

- `dune/fem_bem_toolbox`:
  - `fem_objects`: schemes and model classes for fem problems,
  - `bem_objects`: schemes and model classes for bem problems.
  - `coupling_schemes`: coupling schemes.
  - `data_communication`: main class for gluing of surface/bulk meshes and communication of data between the grids.
  - `load_balancers`: special load balancers for matching surface/bulk grid partitioning.
  - `shared_memory_vectors`: wrapper class for dof vectors using (shared) raw memory blocks.
  - `fluid_models`: schemes and model classes for current and future fluid problems.

  In addition the `dune` folder contains some fixes to other core modules which will not be required once this module have been made compatible with the 2.4 release.
- `misc`: internal DUNE configuration management files
- `src`: internal DUNE configuration management files
- `doc`: documentation for the module
- `misc`: collection of useful bash scripts for installation of the module and downloading of dependent software/libraries – see installation notes below).
  - `mega.build`: bash script using autotools for building entire module and downloading+configuring all dependent software
  - `mega.make`: bash script using CMake for building entire module and downloading+configuring all dependent software (for 2.4 release)
- `case_studies`: source code for all five case studies.

© by the authors, 2017

- `fem_test`: pure fem problem.
- `bem_test`: pure bem problem.
- `coupled_sphere`: coupled bulk fem-surface fem test case.
- `scattering_wave`: coupled bulk fem-surface bem wave scattering problem.
- `charged_droplet`: coupled bulk fem / surface fem + surface bem charged droplet example.

Each case study folder contains three subfolders:

- `source`: the `main.cc` file required to run the simulation.
- `data`: grid files and a file `parameter` containing run time parameters for this example.
- `output`: folder where the simulation output is stored.

B Installation notes

This module is licenced under GPL Version 2. Warning: All the software described in this work has been downloaded, built, run and tested on Linux machines; use on other systems has not been tried, however, if built without `Bem++`– see below – the DUNE-FEM-BEM-TOOLBOX should have the same platform requirements as all other DUNE modules.

To obtain the DUNE-FEM-BEM-TOOLBOX module, the user should clone the `releases/2.3` branch from the DUNE user website by typing the following at the command prompt in their chosen installation directory:

```
git clone -b releases/2.3 \
https://gitlab.dune-project.org/andreas.dedner/dune-fem-bem-toolbox.git
```

Once this is done, to aid the user in the installation of all the remaining software and the compilation and general set-up of all the source code and examples presented, a bash shell script called `mega.build` is included in the "misc" miscellaneous folder. This script also allows setting-up of the module in a reduced state without the `Bem++` library, and thus just using DUNE modules. The scattered wave, *charged* droplet and, of course, bem comparison examples requiring calculation of bem integrals will then be unavailable, however, the FEM comparison and coupled sphere examples will still work, and the droplet example may still be run, although in an *uncharged* state, to give simple oscillations for example.

Before running the script, the user should adapt the `CC` and `CXX` statements at the head of the file to the latest C and C++ compilers available on their system respectively. Additionally, for build efficiency, should a `PYTHON` or `CYTHON` distribution already be available, then typing `export PYTHONPATH=/path/to/my/cython` will avoid the `Bem++` installation process from downloading its own version.

Once the C/C++ compilers have been set, to load all the dependencies and build everything for the toolbox module simply type:

```
./mega.build bem
```

at the comand prompt from within the `misc` directory.

This build script will install and configure some of the ("non-core") dependant DUNE modules at the same folder hierachy level as the DUNE-FEM-BEM-TOOLBOX module itself, so if the user already has DUNE modules around, it is suggested to checkout the DUNE-FEM-BEM-TOOLBOX into an empty installation folder when using this build script.

The `Bem++` software, and any dependencies it builds itself, will be installed within the `mod` folder at the top level within the DUNE-FEM-BEM-TOOLBOX. The user may also notice that `gmsh` software Geuzaine and Remacle [2009] useful for mesh generation (see the running the examples notes above), is also installed by this script, but within the `misc` folder at the same top level.

 © by the authors, 2017

Note that `Bem++` installs most of its dependencies if required (including the DUNE "core" modules requiring the release version 2.3). The dependency on `Cython` can be problematic and might require the user to install that package (version greater or equal to 0.21). separately which on most systems is possible using `pip install --user cython`. Furthermore, `Bem++` is incompatible with certain versions of `Eigen`. `Bem++`'s build system will download a compatible version if `Eigen` is not found on the system. A pre installed `Eigen` can thus lead to problems also with the include paths used while building this module.

There may also be problems installing the `TBB` package on some systems, and if this occurs, it is recommended to preinstall this software too, probably from a tarball downloaded from the TBB website `https://www.threadingbuildingblocks.org`.

`Bem++` will install `Boost` version 1.57 if no version is found on the system during the build process.

To load all the dependencies and build everything *without* `Bem++` simply type

```
./mega.build nobem
```

at the comand prompt within the `misc` directory. Note: `nobem` is also the default setting should no explicit choice be made.

To reload and build just the DUNE dependencies (for dune updates, say) without rebuilding the `Bem++` library type

```
./mega.build bembutnobembuild
```

at the comand prompt within the `misc` directory.

C Known Issues

Inevitably, as with any new software release, there will be a few issues affecting the structure and performance of the software that remain to be sorted out. These are principally related to issues within the supporting software upon which the present module is built, and over which the authors have little, or no, influence. Thus certain accomodations have been made in the present 2.3 release to adapt to these issues.

Issue : No projection of `ALUGrid` surfaces in refinements in parallel

Effect : Convergence testing of coupled sphere example may only be done in serial.

Issue : Memory leaks (either local or within `Bem++`) with destruction of BEM objects for charged drop example when using `Bem++`. Note this does *not* effect the program execution, but merely the internal clean-up on completion.

Effect : Significant internal system output on program completion.

Acknowledgments

The authors gratefully acknowledge the financial assistence of EPSRC under grant number EP/K038060/1 for the support of the second author during his stay at Warwick university, and the suggestions of the reviewers for improving the manuscript.

© by the authors, 2017

References

M. Alkaemper, A. Dedner, R. Kloefkorn, and M. Nolte. The dune-alugrid module. *Archive of Numerical Software*, 4(1), 2016.

Ayachit and Utkarsh. *The ParaView Guide: A Parallel Visualization Application*. Kitware, 2015. ISBN 978-1930934306. URL http://www.paraview.org.

P. Bastian, G. Buse, and O. Sander. Infrastructure for the coupling of dune grids. In *Proceedings of ENUMATH 2009*, pages 107–114, 2010.

E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2), 2012. http://www.cs.sandia.gov/zoltan/.

U. Brink and E.P. Stephan. Adaptive coupling of boundary elements and mixed finite elements for incompressible elasticity. *Numer. Methods Partial Differential Equations*, 17:79–92, 2001.

D. Colton and R. Kress. *Integral Equation Methods in Scattering Theory*. John Wiley & Sons, 1983.

M. Costabel. Symmetric methods for the coupling of finite elements and boundary elements. *Boundary Elements*, 9:411–420, 1987.

Dune. Distributed and unified numerics environment. http://www.dune-project.org, 2012.

dune-fem howto. Howto module for the dune-fem package. http://users.dune-project.org/projects/dune-fem-howto/repository, 2015.

M.. Feischl, T.. Führer, D.. Praetorius, and E. P.. Stephan. Optimal preconditioning for the symmetric and nonsymmetric coupling of adaptive finite elements and boundary elements. *Numerical Methods for Partial Differential Equations*, 2015. ISSN 1098-2426. doi: 10.1002/num.22025. URL http://dx.doi.org/10.1002/num.22025.

D. R. Gaston, C. J. Permann, J. W. Peterson, A. E. Slaughter, D. AndrÅą, Y. Wang, M. P. Short, D. M. Perez, M. R. Tonks, J. Ortensi, L. Zou, and R. C. Martineau. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45 – 54, 2015. ISSN 0306-4549. doi: http://dx.doi.org/10.1016/j.anucene.2014.09.060. URL http://www.sciencedirect.com/science/article/pii/S030645491400543X. Multi-Physics Modelling of {LWR} Static and Transient Behaviour.

G.N. Gatica and N. Heuer. A dual-dual formulation for the coupling of mixed-fem and bem in hyperelasticity. *SIAM J. Num. Anal.*, 38:380–400, 2000.

C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Num. Meth. Eng.*, 79:1309–1331, 2009.

E. Giglio, B. Gervais, J. Rangama, B. Manil, and B.A. Huber. Shape deformations of surface-charged micro-droplets. *Phys. Rev. E*, 77:0363191–7, 2008.

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

M.A. Heald and J.B. Marion. *Classical Electromagnetic Radiation*. Dover, 3 edition, 2012.

M. A. Heroux, R. B. Brightwell, and M. M. Wolf. Bi-modal mpi and mpi+threads computing on scalable multicore systems. In *IJHPCA (Submitted)*, 2011.

R. Hiptmair. Coupling of finite elements and boundary elements in electromagnetic scattering. *SIAM J. Num. Anal.*, 41:919–944, 2003.

 © by the authors, 2017

D. Ibanez, I. Dunn, and M. S. Shephard. Hybrid mpi-thread parallelization of adaptive mesh operations. *Parallel Comput.*, 52(C):133–143, Feb. 2016. ISSN 0167-8191. doi: 10.1016/j.parco.2016.01.003. URL http://dx.doi.org/10.1016/j.parco.2016.01.003.

C. Johnson and J.C. Nedelec. On the coupling of boundary integral and finite element methods. *Math. Comp.*, 35(152):1063–1079, 1980.

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. Kriemann. H-lu factorization on many-core systems. *Computing and Visualization in Science*, 16: 105–117, June 2013. doi: 10.1007/s00791-014-0226-7.

C.C. Lin, E.C. Lawton, J.A. Caliendo, and L.R. Anderson. An iterative finite element-boundary element algorithm. *Comp. Struct.*, 59:899–909, 1996.

S. Meddahi and F.-J. Sayas. A fully discrete bem-fem for the exterior stokes problem in the plane. *SIAM J. Num. Anal.*, 37:2082–2102, 2000.

S. Meddahi and V. Selgas. A mixed-fem and bem coupling for a three-dimensional eddy current problem. *Math. Mod. Num. Anal.*, 37:291–318, 2003.

P. Mund and E.P. Stephan. Adaptive coupling and fast solution of fem-bem equations for parabolic-elliptic interface problems. *Math. Meth. Appl. Sci.*, 20:403–423, 1997.

R. Olson, J. Bentz, R. Kendall, M. Schmidt, and M. Gordon. A novel approach to parallel coupled cluster calculations:ÂĆäâ combining distributed and shared memory techniques for modern cluster based systems. *Journal of Chemical Theory and Computation*, 3(4):1312–1328, 2007. ISSN 1549-9618. doi: 10.1021/ct600366k.

A.J. Radcliffe. A comparison between a symmetric and a non-symmetric galerkin finite element - boundary integral equation coupling for the two-dimensional exterior stokes problem. *Eng. Anal. Bound. Elem.*, 35:959–969, August 2011. DOI: 10.1016/j.enganabound.2011.03.003.

A.J. Radcliffe. Fem-bem coupling for the exterior stokes problem with non-conforming finite elements and an application to small droplet deformation dynamics. *Int. J. Num. Meth. Fluids*, 68:522–536, February 2012. DOI: 10.1002/fld.2518.

A.J. Radcliffe. Non-conforming finite elements for axisymmetric charged droplet deformation dynamics and coulomb explosions. *Int. J. Num. Meth. Fluids*, 71:249–268, January 2013. DOI: 10.1002/fld.3667.

A.J. Radcliffe. Arbitrary lagrangian eulerian simulations of highly electrically charged micro-droplet coulomb explosion deformation pathways. *Int. J. Modeling, Simulation, and Scientific Computing*, 7, February 2016. DOI: 10.1142/S1793962316500161.

T. Ranner and C.M. Elliott. Finite element analysis for a coupled bulk-surface partial differential equation. *IMA Journal of Numerical Analysis*, 33(2):377 – 402, 2013. URL http://eprints.whiterose.ac.uk/77779/.

F.J. Sayas. The validity of johnson–nédélec's bem–fem coupling on polygonal interfaces. *SIAM Journal on Numerical Analysis*, 47(5):3451–3463, 2009. doi: 10.1137/08072334X.

F.J. Sayas. The validity of johnson–nédélec's bem–fem coupling on polygonal interfaces. *SIAM Review*, 55(1):131–146, 2013. doi: 10.1137/120892283.

W. Smigaj, S. Arridge, T. Betcke, J. Phillips, and M. Schweiger. Solving boundary integral problems with bem++. *ACM Trans. Math. Software*, 41:6:1–6:40, 2015.

© by the authors, 2017

E. P. Stephan. *Coupling of Boundary Element Methods and Finite Element Methods*. Encyclopedia of Computational Mechanics. John Wiley and Sons, Ltd., 2004.

TBB. Threading building blocks (intel tbb) 4.4 update 2. https://www.threadingbuildingblocks.org, 2014.

A. Toselli and O.B. Widlund. *Domain Decomposition Methods - Algortihms and Theory*, volume 34 of *Series in Computational Mathematics*. Springer, 2005. ISBN 978-3-540-20696-5.

L. C. Wrobel. *Applications in Thermo-Fluids and Acoustics*, volume 1 of *The Boundary Element Method*. John Wiley and Sons, Ltd., 2002.

 © by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 95–110
DOI: 10.11588/ans.2017.1.27683

The interface for functions in the dune-functions module

Christian Engwer[1], Carsten Gräser[2], Steffen Müthing[3], and Oliver Sander[4]

[1]Universität Münster, Institute for Computational und Applied Mathematics, christian.engwer@uni-muenster.de
[2]Freie Universität Berlin, Institut für Mathematik, graeser@mi.fu-berlin.de
[3]Universität Heidelberg, Institut für Wissenschaftliches Rechnen, steffen.muething@iwr.uni-heidelberg.de
[4]TU Dresden, Institute for Numerical Mathematics, oliver.sander@tu-dresden.de

Received: February 5th, 2016; **final revision:** August 1st, 2016; **published:** March 6th, 2017.

Abstract: The `dune-functions` Dune module introduces a new programmer interface for discrete and non-discrete functions. Unlike the previous interfaces considered in the existing Dune modules, it is based on overloading `operator()`, and returning values by-value. This makes user code much more readable, and allows the incorporation of newer C++ features such as lambda expressions. Run-time polymorphism is implemented not by inheritance, but by type erasure, generalizing the ideas of the `std::function` class from the C++11 standard library. We describe the new interface, show its possibilities, and measure the performance impact of type erasure and return-by-value.

1 Introduction

Ever since its early days, Dune [2, 1] has had a programmer interface for functions. This interface was based on the class `Function`, which basically looked like

C++ code

```
1  template <class Domain, class Range>
2  class Function
3  {
4    public:
5      void evaluate(const Domain& x, Range& y) const;
6  };
```

and is located in the file `dune/common/function.hh`. This class was to serve as a model for duck typing, i.e., any object with its interface would be called a function. A main feature was that the result of a function evaluation was *not* returned as a return value. Rather, it was returned using a by-reference argument of the `evaluate` method. The motivation for this design decision was run-time efficiency. It was believed that returning objects by value would, in practice, lead to too many unnecessary temporary objects and copying operations.

Unfortunately, the old interface lead to user code that was difficult to read in practice. For example, to evaluate the n-th Chebycheff polynomial $T(x) = \cos(n \arccos(x))$ using user methods `my_cos` and `my_arccos` would take several lines of code:

C++ code

```
double tmp1,result;
my_arccos.evaluate(x,tmp1);
my_cos.evaluate(n*tmp1, result);
```

Additionally, C++ compilers have implemented return-value optimization (which can return values by-value without any copying) for a long time, and these implementations have continuously improved in quality. Today, the speed gain obtained by returning result values in a by-reference argument is therefore not worth the clumsy syntax anymore (we demonstrate this in Section 4). We therefore propose a new interface based on `operator()` and returning objects by value. With this new syntax, the Chebycheff example takes the much more readable form

C++ code

```
double result = my_cos(n*my_arccos(x));
```

To implement dynamic polymorphism, the old functions interface uses virtual functions. There is an abstract base class

C++ code

```
template <class Domain, class Range>
class VirtualFunction :
  public Function<const Domain&, Range&>
{
public:
  virtual void evaluate(const Domain& x, Range& y) const = 0;
};
```

in the file `dune/common/function.hh`. User functions that want to make use of run-time polymorphism have to derive from this base class. Calling such a function would incur a small performance penalty [3], which may possibly be avoided by compiler devirtualization [5].

The C++ standard library, however, has opted for a different approach. Instead of deriving different function objects from a common base class, no inheritance is used at all. Instead, any object that implements `operator()` not returning `void` qualifies as a function by duck typing. If a function is passed to another class, the C++ type of the function is a template parameter of the receiving class. If the type is not known at compile time, then the dedicated wrapper class

C++ code

```
namespace std
{
  template< class Range, class... Args >
  class function<Range(Args...)>;
}
```

is used. This class uses type erasure to allow to handle function objects of different C++ types through a common interface. There is a performance penalty for calling functions hidden within a `std::function`, comparable to the penalty for calling virtual functions.

The `dune-functions` module [4] picks up these ideas and extends them. The class template `std::function` works nicely for functions that support only pointwise evaluation. However, in a finite element context, a few additional features are needed. First of all, functions frequently need to supply derivatives as well. Also, for functions defined on a finite element grid, there is typically no single coordinate system of the domain space. Function evaluation in global coordinates is possible, but expensive; such functions are usually evaluated in local coordinates of a given

 © by the authors, 2017

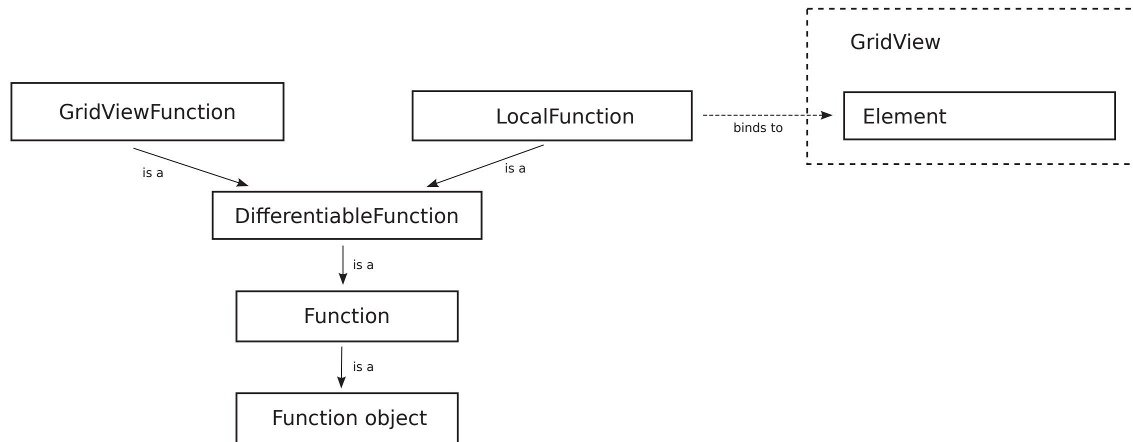


Figure 1: Diagram of the various function interfaces

element. `dune-functions` solves this by introducing `LocalFunction` objects, which can be *bound* to grid elements. When bound to a particular element, they behave just like a `std::function`, but using the local coordinate system of that element.

The paper is structured as follows. The main building blocks for the proposed function interfaces, viz. function objects, concept checks, and type erasure, are introduced in Section 2. Those techniques are then applied to implement the extended interfaces for differentiable functions and grid functions in Section 3. Finally, we present comparative measurements for the current and the proposed interface in Section 4. The results demonstrate that the increased simplicity, flexibility, and expressiveness do not have a negative performance impact.

2 Building blocks for function interfaces

The programmer interface for global functions is given as a set of model classes. These form a conceptual hierarchy (shown in Figure 1), even though no inheritance is involved at all. Implementors of the interface need to write classes that have all the methods and behavior specified by the model classes. This approach results in flexible code. Concept checking is used to produce readable error messages. The following sections explain the individual ideas in detail.

2.1 Function objects and functions

The C++ language proposes a standard way to implement functions. A language construct is called a *function object* if it is an ordinary function, a function pointer, or an object (or reference to an object) of a class providing an `operator()`. In the following we will adopt the common convention to call the latter a *functor*. In the following we will denote a function object as *function* if such a call does not return `void`. In other words, a function `foo` is anything that can appear in an expression of the form

C++ code

```
1 auto y = foo(x);
```

for an argument `x` of suitable type. Examples of such constructs are free functions

C++ code

```
double sinSquared(double x)
{
  return std::sin(x) * std::sin(x);
}
```

lambda expressions

C++ code

```
auto sinSquaredLambda = [](double x){return std::sin(x) * std::sin(x); };
```

functors

C++ code

```
struct SinSquared
{
  double operator()(double x)
  {
    return std::sin(x) * std::sin(x);
  }
};
```

and other things like bind expressions.

All three examples are functions in the above given sense, i.e., they can be called:

C++ code

```
    double a = sinSquared(3.14);       // free function
    double b = sinSquaredLambda(3.14); // lambda expression
    SinSquared sinSquaredObject;
    double c = sinSquaredObject(3.14); // functor
```

Argument and return value do not have to be `double` at all, any type is possible. They can be scalar or vector types, floating point or integer, and even more exotic data like matrices, tensors, and strings.

To pass a function as an argument to a C++ method, the type of that argument must be explicitly stated in the signature of the called method, or it must be a template parameter.

C++ code

```
template <typename F>
foo(F&& f)
{
  std::cout << "Value of f(42): " << f(42) << std::endl;
}
```

Any of the example functions from above can be used as an argument of the method `foo`:

C++ code

```
foo(sinSquared);            // call with a free function
foo(sinSquaredLambda);      // call with a lambda expression
foo(sinSquaredObject);      // call with a functor
```

 © by the authors, 2017

2.2 Concept checks

The calls to F are fast, because the function calls can be inlined. On the other hand, it is not clear from the interface of foo what the signature of F should be. What's worse is that if a function object type with the wrong signature is passed, the compiler error will not occur at the call to foo but only where F is used, which may be less helpful. To prevent this, dune-functions provides light-weight concept checks for the concepts it introduces. For example, the following alternative implementation of foo checks whether F is indeed a function in the sense given above

C++ code

```
template<class F>
void foo(F&& f)
{
  using namespace Dune;
  using namespace Dune::Functions;

  // Get a nice compiler error for inappropriate F
  static_assert(models<Concept::Function<Range(Domain)>, F>(),
      "Type does not model function concept");

  std::cout << "Value of f(42): " << f(42) << std::endl;
}
```

If foo is instantiated with a type that is not a function, say, an int, then a readable error message is produced. For example, for the code

C++ code

```
foo(1);     // The integer 1 is not a function object
```

GCC-4.9.2 prints the error message (file paths and line numbers removed)

```
In instantiation of 'void foo(F&&) [with F = int]':
  required from here
error: static assertion failed: Type does not model function concept
   static_assert(models<Function<Range(Domain)>, F>(),
   ^
```

The provided concept checking facility is based on a list of expressions that a type must support to model a concept. The implementation is based on the techniques proposed by E. Niebler [8] and implemented in the range-v3 library [7]. While the dune-common module provides the general concept checking facility including the models() function that allows to check if a type models a concept, the definitions of the function concepts discussed in this paper are contained in the dune-functions module. This includes the concept Function<Range(Domain)> for simple functions as introduced above, as well as DifferentiableFunction<Range(Domain)>, GridViewFunction<Range(Domain), GridView>, and LocalFunction<Range(Domain), LocalContext> for the extended function interfaces discussed in Section 3.

2.3 Type erasure and std::function

Sometimes, the precise type of a function is not known at compile-time but selected depending on run-time information. This behavior is commonly referred to as *dynamic dispatch*. The classic way to implement this uses inheritance and the virtual keyword: All classes implementing functions must inherit from a common base class, and a pointer to this class is then passed around instead of the function itself.

This approach has a few disadvantages. For example, all functors must live on the heap, and a heap allocation is needed for each function construction. Secondly, in a derived class, the return value of operator() must match the return value used in the base class (weaker rules hold

© by the authors, 2017

for pointer or reference types, which are not of interest to us, though). However, it is frequently convenient to also allow return values that are *convertible* to the return value of the base class. This is not possible in C++. As a third disadvantage, interfaces can only be implemented intrusively, and having one class implement more than a single interface is quite complicated.

The C++ standard library has therefore chosen type erasure over inheritance to implement run-time polymorphism. Starting with C++11, the standard library contains a class [6, 20.8.11]

C++ code

```
namespace std
{
  template< class Range, class... Args >
  class function<Range(Args...)>
}
```

that wraps all functions that map a type `Domain` to a type (convertible to) `Range` behind a single C++ type. A much simplified implementation looks like the following:

C++ code

```
template<class Range, class Domain>
struct function<Range(Domain)>
{
  template<class F>
  function(F&& f) :
    f_(new FunctionWrapper<Range<Domain>, F>(f))
  {}

  Range operator() (Domain x) const
  {
    return f_->operator()(x);
  }

  FunctionWrapperBase<Range<Domain>>* f_;
};
```

The classes `FunctionWrapper` and `FunctionWrapperBase` look like this:

C++ code

```
template<class Range, class Domain>
struct FunctionWrapperBase<Range(Domain)>
{
  virtual Range operator() (Domain x) const=0;
};

template<class Range, class Domain, class F>
struct FunctionWrapper<Range(Domain), F> :
  public FunctionWrapperBase<Range(Domain)>
{
  FunctionWrapper(const F& f) : f_(f) {}

  virtual Range operator() (Domain x) const
  {
    return f_(x);
  }

  F f_;
};
```

Given two types `Domain` and `Range`, any function object that accepts a `Domain` as argument and returns something convertible to `Range` can be stored in a `std::function<Range(Domain)>`. For example, reusing the three implementations of $\sin^2(x)$ from Section 2.1, one can write

 © by the authors, 2017

C++ code

```
std::function<double(double)> polymorphicF;
polymorphicF = sinSquared;          // assign a free function
polymorphicF = sinSquaredLambda;    // assign a lambda expression
polymorphicF = SinSquared();        // assign a functor
double a = polymorphicF(3.14); // evaluate
```

Note how different C++ constructs are all assigned to the same object. One can even use

C++ code

```
polymorphicF = [](double x) -> int { return floor(x); };
   // okay: int can be converted to double
```

but not

C++ code

```
polymorphicF = [](double x) -> std::complex<double>
   { return std::complex<double>(x,0); };
   // error: std::complex<double> cannot be converted to double
```

Looking at the implementation of `std::function`, one can see that virtual functions are used *internally*, but it is completely hidden to the outside. The copy constructor accepts any type as argument. In a full implementation the same is true for the move constructor, and the copy and move assignment operators. For each function type `F`, an object of type `FunctionWrapper<Range(Domain),F>` is constructed, which inherits from the abstract base class `FunctionWrapperBase<Range(Domain)>`.

Considering the implementation of `std::function` as described here, one may not expect any run-time gains for type erasure over virtual methods. While `std::function` itself does not have any virtual methods, each call to `operator()` does get routed through a virtual function. Additionally, each call to the copy constructor or assignment operator invokes a heap allocation. The virtual function call is the price for run-time polymorphism. It can only be avoided in some cases using smart compiler devirtualization.

To alleviate the cost of the heap allocation, `std::function` implements a technique called small object optimization. In addition to the pointer to `FunctionWrapper`, a `std::function` stores a small amount of raw memory. If the function is small enough to fit into this memory, it is stored there. Only in the case that more memory is needed, a heap allocation is performed. Small object optimization is therefore a trade-off between run-time and space requirements. `std::function` needs more memory with it, but is faster *for small objects*.

Small object optimization is not restricted to type erasure, and can in principle be used wherever heap allocations are involved. However, with an inheritance approach this nontrivial optimization would have to be exposed to the user code, while all its details are hidden from the user in a type erasure context.

While we rely on `std::function` as a type erasure class for global functions, this is not sufficient to represent extended function interfaces as discussed below. To this end `dune-functions` provides utility functionality to implement new type erasure classes with minimal effort, hiding, e.g., the details of small objects optimization. This can be used to implement type erasure for extended function interfaces that go beyond the ones provided by `dune-functions`. Since these utilities are not function-specific, they can also support the implementation of type-erased interfaces in other contexts. Similar functionality is, e.g., provided by the *poly* library (which is part of the *Adobe Source Libraries* [9]), and the *boost type erasure* library [10].

© by the authors, 2017

3 Extended function interfaces

The techniques discussed until now allow to model functions

$$f : \mathcal{D} \to \mathcal{R} \tag{1}$$

between a domain $\mathcal{D}$ and a range $\mathcal{R}$ by interfaces using either static or dynamic dispatch. In addition to this, numerical applications often require to model further properties of a function like differentiability or the fact that it is naturally defined locally on grid elements. In this section we describe how this is achieved in the `dune-functions` module using the techniques described above.

3.1 Differentiable functions

The extension of the concept for a function (1) to a differentiable function requires to also provide access to its derivative

$$Df : \mathcal{D} \to L(\mathcal{D}, \mathcal{R}) \tag{2}$$

where, in the simplest case, $L(\mathcal{D}, \mathcal{R})$ is the set of linear maps from the affine hull of $\mathcal{D}$ to $\mathcal{R}$.

Example 1. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ the derivative Df maps each vector $x \in \mathcal{D} = \mathbb{R}^n$ to a linear map $Df(x)$ from $\mathcal{D} = \mathbb{R}^n$ to $\mathcal{R} = \mathbb{R}^m$. Since we can identify any linear map $M : \mathbb{R}^n \to \mathbb{R}^m$ with a matrix $M \in \mathbb{R}^{m \times n}$ such that the value $M(y)$ of M at $y \in \mathbb{R}^n$ is given by the matrix vector product $M(y) = My \in \mathbb{R}^m$ we have identified $L(\mathcal{D}, \mathcal{R})$ with $\mathbb{R}^{m \times n}$. Hence Df is itself a function mapping vectors from $\mathbb{R}^n$ to matrices from $\mathbb{R}^{m \times n}$. Notice that $Df(x) \in \mathbb{R}^{m \times n}$ is commonly referred to as the *Jacobian matrix* of f at x.

To provide access to derivatives, the `dune-functions` module extends the ideas from the previous section in a natural way. A C++ construct is a differentiable function if, in addition to having `operator()` as described above, there is a free method `derivative` that returns a function that implements the derivative. The typical way to do this will be a friend function as illustrated in the class template `Polynomial`:

C++ code

```
1  template<class K>
2  class Polynomial
3  {
4  public:
5
6    Polynomial() = default;
7    Polynomial(const Polynomial& other) = default;
8    Polynomial(Polynomial&& other) = default;
9
10   Polynomial(std::initializer_list<double> coefficients) :
11     coefficients_(coefficients) {}
12
13   Polynomial(std::vector<K>&& coefficients) :
14       coefficients_(std::move(coefficients)) {}
15
16   Polynomial(const std::vector<K>& coefficients) :
17     coefficients_(coefficients) {}
18
19   const std::vector<K>& coefficients() const
20   { return coefficients_; }
21
22   K operator() (const K& x) const
23   {
24     auto y = K(0);
25     for (size_t i=0; i<coefficients_.size(); ++i)
26       y += coefficients_[i] * std::pow(x, i);
27     return y;
```

 © by the authors, 2017

```
28    }
29
30    friend Polynomial derivative(const Polynomial& p)
31    {
32      std::vector<K> dpCoefficients(p.coefficients().size()-1);
33      for (size_t i=1; i<p.coefficients_.size(); ++i)
34        dpCoefficients[i-1] = p.coefficients()[i]*i;
35      return Polynomial(std::move(dpCoefficients));
36    }
37
38  private:
39    std::vector<K> coefficients_;
40  };
```

To use this class, write

C++ code

```
1  auto f = Polynomial<double>({1, 2, 3});
2  double a = f(3.14);
3  double b = derivative(f)(3.14);
```

Note, however, that the `derivative` method may be expensive, because it needs to compute the entire derivative function. It is therefore usually preferable to call it only once and to store the derivative function separately.

C++ code

```
1  auto df = derivative(f);
2  double b = df(3.14);
```

Functions supporting these operations are described by `Concept::DifferentiableFunction`, provided in the `dune-functions` module.

When combining differentiable functions and dynamic polymorphism, the `std::function` class cannot be used as is, because it does not provide access to the `derivative` method. However, it can serve as inspiration for more general type erasure wrappers. The `dune-functions` module provides the class

C++ code

```
1  template<class Signature,
2           template<class> class DerivativeTraits=DefaultDerivativeTraits,
3           size_t bufferSize=56>
4  class DifferentiableFunction;
```

in the file `dune/functions/common/differentiablefunction.hh`. Partially, it is a reimplementation of `std::function`. The first template argument of `DifferentiableFunction` is equivalent to the template argument `Range(Args...)` of `std::function`, and `DifferentiableFunction` implements a method

C++ code

```
1  Range operator() (const Domain& x) const;
```

This method works essentially like the one in `std::function`, despite the fact that its argument type is fixed to `const Domain&` instead of `Domain`, because arguments of a "mathematical function" are always immutable. Besides this, it also implements a free method

C++ code

```
1  friend DerivativeInterface derivative(const DifferentiableFunction& t);
```

that wraps the corresponding method of the function implementation. It allows to call the `derivative` method for objects whose precise type is determined only at run-time:

C++ code

```
DifferentiableFunction<double(double)> polymorphicF;
polymorphicF = Polynomial<double>({1, 2, 3});
double a = polymorphicF(3.14);
auto polymorphicDF = derivative(polymorphicF);
double b = polymorphicDF(3.14);
```

While the domain of a derivative is $\mathcal{D}$, the same as the one of the original function, its range is $L(\mathcal{D}, \mathcal{R})$. Unfortunately, it is not feasible to always infer the best C++ type for objects from $L(\mathcal{D}, \mathcal{R})$. To deal with this, `dune-functions` offers the `DerivativeTraits` mechanism that maps the signature of a function to the range type of its derivative. The line

C++ code

```
using DerivativeRange = DerivativeTraits<Range(Domain)>::Range;
```

shows how to access the type that should be used to represent elements of $L(\mathcal{D}, \mathcal{R})$. The template `DefaultDerivativeTraits` is specialized for common combinations of Dune matrix and vector types, and provides reasonable defaults for the derivative ranges. However, it is also possible to change this by passing a custom `DerivativeTraits` template to the interface classes, e.g., to allow optimized application-specific matrix and vector types or use suitable representations for other or generalized derivative concepts.

Currently the design of `DifferentiableFunction` differs from `std:function` in that it only considers a single argument, but this can be vector valued.

3.2 GridView functions and local functions

A very important class of functions in any finite element application are discrete functions, i.e., functions that are defined piecewisely with respect to a given grid. Here, a grid covering a domain $\mathcal{D}$ is a decomposition into a set of subdomains in the sense that

$$\overline{\mathcal{D}} = \bigcup_{e \in \mathcal{G}} \bar{e}.$$

The subdomains $e \in \mathcal{G}$ are called *grid elements* or *cells* of the grid. In many applications with $\Omega \subset \mathcal{R}^d$ those elements are disjoint, open, nonempty d-dimensional polyhedra, possibly with curved boundaries. In Dune terminology, the k-dimensional faces of those polyhedra are called *grid entities* of codimension $d - k$. Hence the elements are the entities of codimension 0 while vertices, edges, and facets of those polyhedra are grid entities of codimension $k = d$, $k = d-1$, and $k = 1$, respectively. For each element $e \in \mathcal{G}$ we assume that there is a so-called *reference element* $\hat{e}$ and a bijective parametrization $\Phi_e : \hat{e} \to \bar{e}$. We call $\xi \in \hat{e}$ the local coordinate of $x \in \overline{\mathcal{D}}$ with respect to an element $e \in \mathcal{G}$ if $x \in \bar{e}$ and $\Phi_e(\xi) = x$.

For a given grid $\mathcal{G}$ on $\mathcal{D}$ a function $f : \mathcal{D} \to \mathcal{R}$ is called a discrete function if it has a natural piecewise definition with respect to the decomposition. Such functions are typically too expensive to evaluate in global coordinates, e.g., at points $x \in \mathcal{D}$ directly. Luckily this is hardly ever necessary. Instead one often knows an element e and local coordinates $\xi \in \bar{\hat{e}}$ of x with respect to e that allow to evaluate $f(x) = f(\Phi_e(\xi))$ cheaply. Formally, this means that we have localized versions

$$f_e = f \circ \Phi_e : \hat{e} \to \mathcal{R}, \qquad e \text{ is element of the grid.}$$

To support this kind of function evaluation, Dune has provided interfaces in the style of

 © by the authors, 2017

C++ code

```
void evaluateLocal(Codim<0>::Entity element,
                   const LocalCoordinates& x,
                   Range& y);
```

Given an element `element` and a local coordinate `x`, such a method would evaluate the function at the given position, and return the result in the third argument `y`. This approach is currently used, e.g., in the grid function interfaces of the discretization modules `dune-pdelab` and `dune-fufem`.

There are several disadvantages to this approach. First, we have argued earlier that return-by-value is preferable to return-by-reference. Hence, an obvious improvement would be to use

C++ code

```
Range operator()(Codim<0>::Entity element, cost LocalCoordinates& x);
```

instead of the `evaluateLocal` method. However, there is a second disadvantage. In a typical access pattern in a finite element implementation, a function evaluation on a given element is likely to be followed by evaluations on the same element. For example, think of a quadrature loop that evaluates a coefficient function at all quadrature points of a given element. Function evaluation in local coordinates of an element can involve some setup code that depends on the element but not on the local coordinate `x`. This could be, e.g., pre-fetching of those coefficient vector entries that are needed to evaluate a finite element function on the given element, or retrieving the associated shape functions.

In the approaches described so far in this section, this setup code is executed again and again for each evaluation on the same element. To avoid this we propose the following usage pattern instead:

C++ code

```
auto localF = localFunction(f);
localF.bind(element);
auto y = localF(xLocal);           // evaluate f in element-local coordinates
```

Here we first obtain a *local function*, which represents the restriction of `f` to a single element. This function is then bound to a specific element using the method

C++ code

```
void bind(Codim<0>::Entity element);
```

This is the place for the function to perform any required setup procedures. Afterwards the local function can be evaluated using the interface described above, but now using local coordinates with respect to the element that the local function is bound to. The same localized function object can be used for other elements by calling `bind` with a different argument. Functions supporting these operations are called *grid view functions*, and described by `Concept::GridViewFunction` The local functions are described by `Concept::LocalFunction`. Both concepts are provided in the `dune-functions` module.

Since functions in a finite element context are usually at least piecewise differentiable, grid view functions as well as local functions provid the full interface of differentiable functions as outlined in Section 3.1. To completely grasp the semantics of the interface, observe that strictly speaking localization does not commute with taking the derivative. Formally, a localized version of the derivative is given by

$$(Df)_e : \hat{e} \to L(\mathcal{D},\mathcal{R}), \qquad (Df)_e = (Df) \circ \Phi_e. \tag{3}$$

In contrast, the derivative of a localized function is given by

$$D(f_e) : \hat{e} \to L(\hat{e},\mathcal{R}), \qquad D(f_e) = ((Df) \circ \Phi_e) \cdot D\Phi_e.$$

However, in the dune-functions implementation, the derivative of a local function does by convention always return values *in global coordinates*. Hence, the functions dfe1 and dfe2 obtained by

C++ code

```
1  auto df = derivative(f);
2  auto dfe1 = localFunction(df);
3  dfe1.bind(element);
4
5  auto fe = localFunction(f);
6  fe.bind(element);
7  auto dfe2 = derivative(fe);
```

both *behave the same*, implementing $(Df)_e$ as in (3). This is motivated by the fact that $D(f_e)$ is hardly ever used in applications, whereas $(Df)_e$ is needed frequently. To express this mild inconsistency in the interface, a local function uses a special DerivativeTraits implementation that forwards the derivative range to the one of the corresponding global function.

Again, type erasure classes allow to use grid view and local functions in a polymorphic way. The class

C++ code

```
1  template<class Signature,
2           class GridView,
3           template<class> class DerivativeTraits=DefaultDerivativeTraits,
4           size_t bufferSize=56>
5  class GridViewFunction;
```

stores any function that models the GridViewFunction concept with given signature and grid view type. Similarly, functions modeling the LocalFunction concept can be stored in the class

C++ code

```
1  template<class Signature,
2           class Element,
3           template<class> class DerivativeTraits=DefaultDerivativeTraits,
4           size_t bufferSize=56>
5  class LocalFunction;
```

These type erasure classes can be used in combination:

C++ code

```
1  GridViewFunction<double(GlobalCoordinate), GridView> polymorphicF;
2  polymorphicF = f;
3  auto polymorphicLocalF = localFunction(polymorphicF);
4  polymorphicLocalF.bind(element);
5  LocalCoordinate xLocal = ... ;
6  auto y = polymorphicLocalF(xLocal);
```

Notice that, as described above, the DerivativeTraits used in polymorphicLocalF are not the same as the ones used by polymorphicF. Instead, they are a special implementation forwarding to the global derivative range even for the domain type LocalCoordinate.

4 Performance measurements

In this last chapter we investigate how the interface design for functions in Dune influences the run-time efficiency. Two particular design choices are expected to be critical regarding execution speed: (i) returning the results of function evaluations by value involves temporary objects and copying unless the compiler is smart enough to remove those using return-value-optimization. In the old interface, such copying could not occur by construction, (ii) using type erasure instead

 © by the authors, 2017

of virtual functions for dynamic polymorphism. While there are fewer reasons to believe that this may cause changes in execution time, it is still worthwhile to check empirically.

As a benchmark we have implemented a small C++ program that computes the integral

$$I(f) := \int_0^1 f(x)\,dx$$

for different integrands, using a standard composite mid-point rule. We chose this problem because it is very simple, but still an actual numerical algorithm. More importantly, most of the time is spent evaluating the integrand function. Finally, hardly any main memory is needed, and hence memory bandwidth limitations will not influence the measurements. We have deliberately omitted tests for derivatives and piecewise functions. As these use return-by-value and type erasure in much the same way as function evaluation does, we do not expect much additional information from such additional tests.

The example code is a pure C++11 implementation with no reference to DUNE. The relevant interfaces from DUNE are so short that it was considered preferable to copy them into the benchmark code to allow easier building. The code is available in a single file attached to this pdf document, via the icon in the margin.

To check the influence of return types with different size we used integrands of the form

$$f : \mathbb{R} \to \mathbb{R}^N, \qquad f(x)_i = x + i - 1, \qquad i = 1, \dots, N,$$

for various sizes N. This special choice was made to keep the computational work done inside of the function to a minimum while avoiding compiler optimizations that replace the function call by a compile-time expression. The test was performed with $n = \lfloor 10^8/N \rfloor$ subintervals for the composite mid-point rule leading to n function evaluations, such that the timings are directly comparable for different values of N.

For the test we implemented four variants of function evaluation:

(a) Return-by-value with static dispatch using plain `operator()`,

(b) Return via reference with static dispatch using `evaluate()`,

(c) Return-by-value with dynamic dispatch using `std::function::operator()`,

(d) Return via reference with dynamic dispatch using `VirtualFunction::evaluate()`, as in the introduction.

The test was performed with $N = 1, \dots, 16$ components for the function range, using `double` to implement the components. We used GCC-4.9.2 and Clang-3.6 as compilers, as provided by the Linux distribution Ubuntu 15.04. To avoid cache effects and to eliminate outliers we did a warm-up run before each measured test run and selected the minimum of four subsequent runs for all presented values.

Figure 2.A shows the execution time in milliseconds over N when compiling with GCC-4.9 and the compiler options `-std=c++11 -O3 -funroll-loops`. One can observe that the execution time is the same for variants (a) and (b) and all values of N. We conclude that for static dispatch there is no run-time overhead when using return-by-value, or, more precisely, that the compiler is able to optimize away any overhead. Comparing the dynamic dispatch variants (c) and (d) we see that for small values of N there is an overhead for return-by-value with type erasure compared to the classic approach using inheritance and virtual functions. This is somewhat surprising since pure return-by-value does not impose an overhead, and dynamic dispatch happens for both variants.

Guessing that the compiler is not able to optimize the nested function calls in the type erasure interface class `std::function` to full extent, we repeated the tests using *profile guided optimization.*

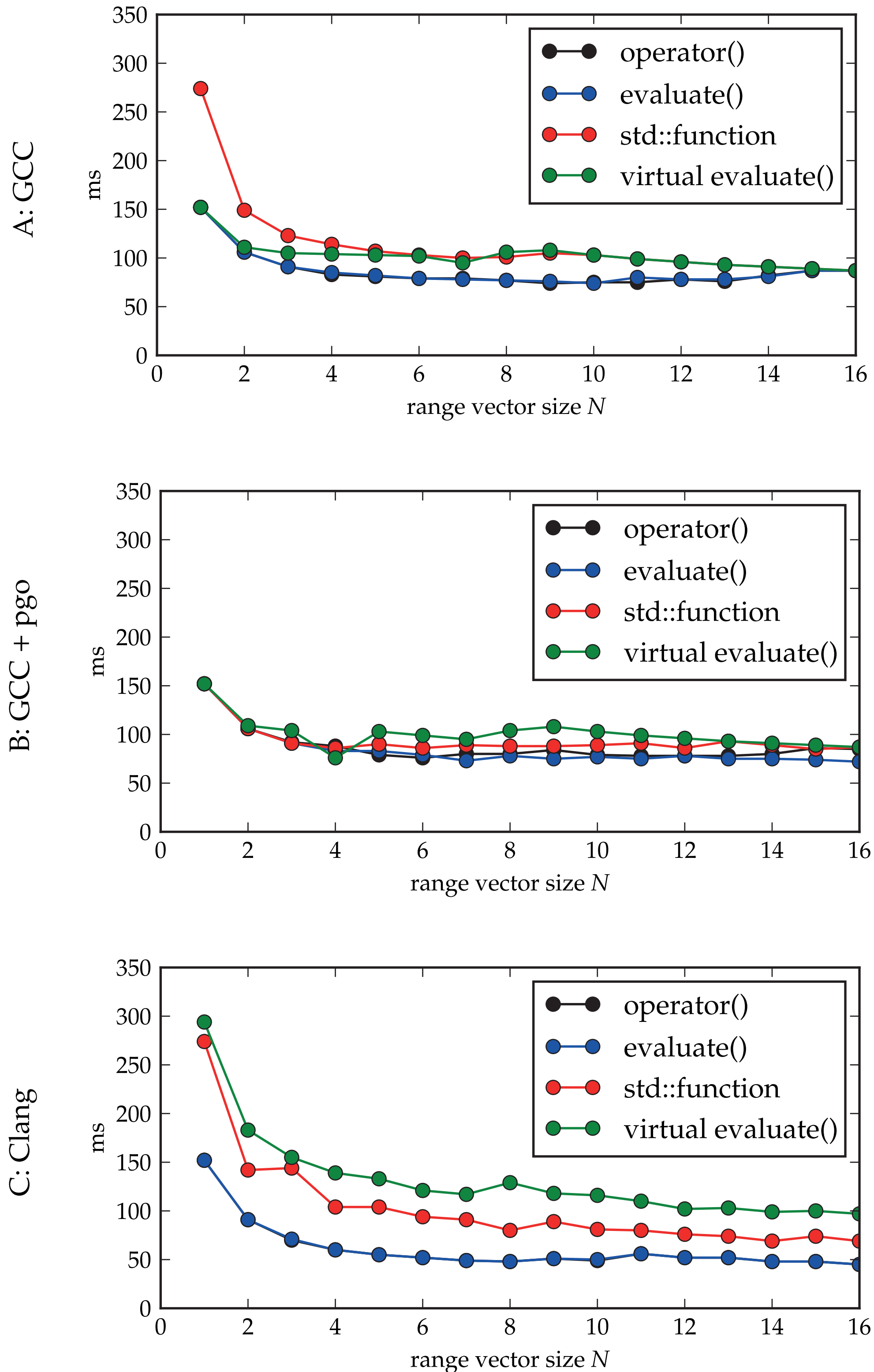


Figure 2: Timings for $\lfloor 10^8/N \rfloor$ function calls over varying vector size N using (A) GCC, (B) GCC with profile-guided optimization, and (C) Clang.

 © by the authors, 2017

To this end the code was first compiled using the additional option `-fprofile-generate`. When running the obtained program once, it generates statistics on method calls that are used by subsequent compilations with the additional option `-fprofile-use` to guide the optimizer. The results depicted in Figure 2.B show that the compiler is now able to generate code that performs equally well for variant (c) and (d). In fact variant (c) is sometime even slightly faster.

Finally, Figure 2.C shows results for Clang-3.6 and the compiler options `-std=c++11 -O3 -funroll-loops`. Again variants (a) and (b) show identical results. In contrast, variant (c) using `std::function` is now clearly superior compared to variant (d). Note that we only used general-purpose optimization options and that this result did not require fine-tuning with more specialized compiler flags.

5 Conclusion

We have presented a new interface for functions in Dune, which is implemented in the new `dune-functions` module. The interface follows the ideas of function objects and `std::function` from the C++ standard library, and generalises these concepts to allow for differentiable functions and discrete grid functions. For run-time polymorphism we offer corresponding type erasure classes similar to `std::function`. The performance of these new interfaces was compared to existing interfaces in Dune. When using the optimization features of modern compilers, the proposed new interfaces are at least as efficient as the old ones, while being much easier to read and use.

References

[1] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for adaptive and parallel scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for adaptive and parallel scientific computing. Part I: Abstract framework. *Computing*, 82(2–3):103–119, 2008.

[3] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 306–323. ACM, 1996.

[4] C. Engwer, C. Gräser, S. Müthing, and O. Sander. Dune-functions module. `http://www.dune-project.org/modules/dune-functions`.

[5] J. Hubička. Devirtualization in C++. online blog, `http://hubicka.blogspot.de/2014/01/devirtualization-in-c-part-1.html`, 2014. (at least) seven parts, last checked on Dec. 8. 2015.

[6] International Organization for Standardization. ISO/IEC 14882:2011 Programming Language C++, 9 2011.

[7] E. Niebler. Range-v3 library. `https://github.com/ericniebler/range-v3`.

[8] E. Niebler. Concept checking in C++11. online blog, `http://ericniebler.com/2013/11/23/concept-checking-in-c11`, 2013. last checked on Dec. 8. 2015.

[9] S. Parent, M. Marcus, and F. Brereton. Adobe source libraries. `http://stlab.adobe.com/`.

[10] S. Watanabe. Boost type erasure library. `http://www.boost.org/doc/libs/release/libs/type_erasure/`.

© by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 111–128
DOI: 10.11588/ans.2017.1.27719

The Dune-DPG library for solving PDEs with Discontinuous Petrov–Galerkin finite elements

Felix Gruber[1], Angela Klewinghaus[1], and Olga Mula[2]

[1]IGPM, RWTH Aachen, Templergraben 55, 52056 Aachen, Germany
[2]Université Paris-Dauphine, PSL Research University, CNRS, UMR 7534, CEREMADE, 75016 Paris, France

Received: February 7th, 2016; **final revision:** November 7th, 2016; **published:** March 6th, 2017.

Abstract: In the numerical solution of partial differential equations (PDEs), a central question is the one of building variational formulations that are inf-sup stable not only at the infinite-dimensional level, but also at the finite-dimensional one. These properties are important since they represent the rigorous foundations for a posteriori error control and the development of adaptive strategies. The essential difficulty lies in finding *systematic* procedures to build variational formulations for which these desirable stability properties are (i) provable at the theoretical level while (ii) the approach remains implementable in practice and (iii) its computational complexity does not explode with the problem size. In this framework, the so-called Discontinuous Petrov–Galerkin (DPG) concept seems a promising approach to enlarge the scope of problems beyond second order elliptic PDEs for which this is possible. In the context of DPG, the result for the elliptic case was proven by Gopalakrishnan and Qiu [2014] and requires a p-enriched test space. Recently, the same type of result has been proven by Broersen et al. [2015] for certain classes of linear transport problems using an appropriate hp-enrichment to build the finite dimensional test space. In the light of this new result, we present Dune-DPG, a C++ library which allows to implement the test spaces introduced in Broersen et al. [2015]. The library is built upon the multi-purpose finite element package Dune (see Blatt et al. [2016]). In this paper, we present the current version 0.2.1 of Dune-DPG which has so far been tested only for elliptic and transport problems. An example of use via a simple transport equation is described. We conclude outlining future work and applications to more complex problems. Dune-DPG is licensed under the GPL 2 with runtime exception and a source code tarball is available together with this paper.

1 Introduction

General context: Let Ω be a domain of $\mathbb{R}^d$ $(d \geq 1)$ and $\mathbb{U}$, $\mathbb{V}$ two Hilbert spaces defined over Ω and endowed with norms $\|\cdot\|_{\mathbb{U}}$ and $\|\cdot\|_{\mathbb{V}}$, respectively. The normed dual of $\mathbb{V}$, denoted $\mathbb{V}'$, is endowed with the norm

$$\|\ell\|_{\mathbb{V}'} := \sup_{v\in\mathbb{V}} \frac{|\ell(v)|}{\|v\|_{\mathbb{V}}}, \quad \forall \ell \in \mathbb{V}'.$$

Let $\mathcal{B}\colon \mathbb{U} \to \mathbb{V}$ be a boundedly invertible linear operator and let $b\colon \mathbb{U}\times\mathbb{V} \to \mathbb{R}$ be its associated continuous bilinear form defined by $b(w,v) = (\mathcal{B}w)(v)$, $\forall (w,v) \in \mathbb{U}\times\mathbb{V}$. We consider the operator equation

$$\begin{aligned} &\text{Given } f \in \mathbb{V}', \text{ find } u \in \mathbb{U} \text{ s. t.} \\ &\mathcal{B}u = f, \end{aligned} \tag{1}$$

or, equivalently, the variational problem

$$\begin{aligned} &\text{Given } f \in \mathbb{V}', \text{ find } u \in \mathbb{U} \text{ s. t.} \\ &b(u,v) = f(v), \quad \forall v \in \mathbb{V}. \end{aligned} \tag{2}$$

Let $0 < \gamma \leq 1$ be a lower bound for the (infinite-dimensional) inf-sup constant

$$\inf_{w\in\mathbb{U}} \sup_{v\in\mathbb{V}} \frac{b(w,v)}{\|w\|_{\mathbb{U}}\|v\|_{\mathbb{V}}} \geq \gamma > 0.$$

Since $\mathcal{B}$ is invertible, problem (1) admits a unique solution $u \in \mathbb{U}$ and for any approximation $\bar{u} \in \mathbb{U}$ of u,

$$\|\mathcal{B}\|^{-1}_{L(\mathbb{U},\mathbb{V}')}\|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'} \leq \|u - \bar{u}\|_{\mathbb{U}} \leq \gamma^{-1}\|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'}. \tag{3}$$

From (3), it follows that the error $\|u - \bar{u}\|_{\mathbb{U}}$ is equivalent to the residual $\|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'}$. The residual contains known quantities and its estimation on an appropriate finite-dimensional space opens the door to rigorously founded a posteriori concepts. However, note that the information that the estimator can give is only meaningful when the variational formulation is well-conditioned, i. e., for $\|\mathcal{B}\|_{L(\mathbb{U},\mathbb{V}')}$ and γ being as close to one as possible. Assuming that we have this property of well-conditioning, a crucial point is that this needs to be inherited at the finite-dimensional level. This issue has been well explored for standard Galerkin methods (i. e. when $\mathbb{U} = \mathbb{V}$) and allows to appropriately address most parabolic and second order elliptic problems with a wide variety of finite element methods. However, there are problems for which standard Galerkin methods do not lead to a stable discretization (one example being transport-dominated PDEs). In this context, one can move to a Petrov-Galerkin framework where the main guiding ideas are:

- It is possible to find a test space $\mathbb{V}$ and a norm for $\mathbb{V}$ which yield $\gamma = 1$. $\mathbb{V}$ and its norm depend on the specific PDE and might not coincide with $\mathbb{U}$.
- For a given finite-dimensional trial space $\mathbb{U}_H$, there exists a corresponding optimal test space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ such that the discrete inf-sup condition

$$\inf_{w_H\in\mathbb{U}_H} \sup_{v\in\mathbb{V}^{\text{opt}}(\mathbb{U}_H)} \frac{b(w_H,v)}{\|w_H\|_{\mathbb{U}}\|v\|_{\mathbb{V}}} \geq \gamma$$

  holds with the same constant γ as the infinite-dimensional one. For more details on this, see Section 2.1.

Since, in general, even the approximate computation of the optimal test space requires the solution of global problems, there are essentially two ways to make the computation affordable. One is the introduction of a mixed formulation which avoids the computation of the optimal test spaces and only uses them indirectly. This approach was used in Dahmen et al. [2012] to construct a general adaptive scheme when $\mathbb{U} = L_2(\Omega)$ and to show convergence under certain abstract conditions (which have to be verified for concrete applications). It was also employed in the context of reduced-basis construction for transport-dominated problems (see Dahmen et al. [2014]).

The other way is to make computations affordable by localization so that the optimal test spaces can be computed by solving local problems. This is the approach taken in the DPG methodology, initiated and developed mainly by L. Demkowicz and J. Gopalakrishnan (see e. g. Demkowicz

and Gopalakrishnan [2011], Gopalakrishnan and Qiu [2014], Demkowicz and Gopalakrishnan [2015]). Since, in general, the local problems to find $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ are still infinite dimensional, the DPG method approximates the exact optimal test functions in the context of a discontinuous Petrov–Galerkin formulation. As a result, $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ is replaced in practical computations by a finite dimensional space called "near-optimal test-space", which we will denote $\mathbb{V}^{\text{n.opt}}$. This is the space which is eventually used in the solution of the discrete problem. For fairly broad classes of PDEs, there exist abstract results showing that the inf-sup stability of the discrete problem with $\mathbb{V}^{\text{n.opt}}$ is preserved provided that $\mathbb{V}^{\text{n.opt}}$ is close enough to the true optimal test-space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ in a certain abstract sense. We refer to Roberts et al. [2014] and also the concept of delta proximality introduced in Dahmen et al. [2012] for results in this respect. Deriving more quantifiable conditions on the form of $\mathbb{V}^{\text{n.opt}}$ is a more involved task which depends on the PDE. It is nevertheless important since it helps to quantify the complexity of the approach in more specific terms. This type of result was first proven for second order elliptic problems by Gopalakrishnan and Qiu [2014]. It requires a p-enrichment strategy (with respect to the polynomial degree of $\mathbb{U}_H$) to build $\mathbb{V}^{\text{n.opt}}$. Recently, the same type of result has been derived by Broersen et al. [2015] for certain classes of linear transport problems. It is proven there that the use of a certain hp-enrichment in the construction of $\mathbb{V}^{\text{n.opt}}$ guarantees a sufficiently good near-optimal test space. Beyond these theoretically backed-up cases, we also note that numerical evidence illustrates the good stability properties of DPG for a larger spectrum of PDEs than elliptic and transport problems. Without being exhaustive, we can find works on convection–diffusion [Broersen and Stevenson, 2015, Niemi et al., 2013], elasticity and Stokes [Carstensen et al., 2014, Roberts et al., 2014], Maxwell equations [Carstensen et al., 2016] and the Helmholtz equation [Demkowicz et al., 2012]. Also, we note that the idea of hp-enrichment to build $\mathbb{V}^{\text{n.opt}}$ was explored in numerical experiments about convection-diffusion in Niemi et al. [2013].

Motivations and contributions of the paper: In this paper, we explain the construction of the Dune-DPG library which aims at solving different types of PDEs with a DPG variational formulation. The main guidelines for its construction have been:

- to give the user as much liberty as possible in the nature of the problem to be addressed, in the nature of the test and trial spaces to be used and in the geometry and mesh refinement,
- to endow the code with a modular structure in order to ensure an easy access to low-level functionalities for which a fine control is required while keeping a high-level view for the rest of the code.

In the light of the recent results by Broersen et al. [2015] and in order to have a rigorous back-up on the stability of the calculations for the largest possible scope of problems, a special emphasis has been put on giving the possibility to use h and p refined spaces for the construction of the near-optimal test space $\mathbb{V}^{\text{n.opt}}$. This point is actually the main novelty that the present library offers with respect to other existing DPG libraries like *Camellia* (see Roberts [2014]). It is nevertheless significant in the sense that this capability seems well suited to investigate novel numerical schemes with rigorous error bounds for more elaborate transport based PDEs such as kinetic problems. A work in this direction is currently ongoing and will be subject of a future publication.

Dune-DPG is licensed under the GPL 2 with runtime exception and its latest version (0.2.1) is released together with this paper[1]. The library is based upon the multi-purpose finite element package Dune (see, e. g., Bastian et al. [2008]). In particular, we use version 2.4.1 [Blatt et al., 2016] of the core modules Dune-Grid [Bastian et al., 2008], Dune-ISTL [Blatt and Bastian, 2006], Dune-Geometry, Dune-Functions [Engwer et al., 2015], Dune-LocalFunctions and Dune-TypeTree. As a consequence, for the users of Dune, the library represents a relatively simple way to experiment with basic DPG concepts.

[1] The whole Git history of Dune-DPG can be found at `https://gitlab.dune-project.org/felix.gruber/dune-dpg`. We try to keep the master branch compatible to the current development branches of the Dune core modules.

Finally, we would like to point out that Dune-DPG is still under active development and the present release comes with a couple of limitations which will be fixed in future versions. These are:

- the current implementation is limited to problems with constant coefficients,
- parallelization of the code has not been explored,
- the library has been tested so far in elliptic and pure transport problems but, as already brought up, it is currently being used to study more involved PDEs.

Layout of the paper: To show how the library works, the paper is organized as follows: In Section 2, we summarize the mathematical concepts of DPG that are relevant to understand the library. As an example, we explain at the end of this section how the ideas can be applied to a simple transport problem following the theory of Broersen et al. [2015]. Then, in Section 3, we present the different building blocks that form the library. We explain how they interact and how they make use of some features of the Dune framework upon which our library is built. Finally, in Section 4, we validate Dune-DPG by giving concrete results related to the solution of a simple transport problem. Some performance results are also given.

2 Theoretical Foundations for DPG

As already brought up in the introduction, the DPG concept was initiated and developed mainly by L. Demkowicz and J. Gopalakrishnan (see e. g. Demkowicz and Gopalakrishnan [2011], Gopalakrishnan and Qiu [2014]). Other relevant results concerning theoretical foundations are Broersen and Stevenson [2014, 2015] and, more recently, Broersen et al. [2015]. The strategy followed in DPG to contrive stable variational formulations is based on the concept of *optimal test spaces* and their practical approximation through the solution of *local* problems in the context of a discontinuous Petrov–Galerkin variational formulation. The two following sections explain more in detail these two fundamental ideas.

2.1 The concepts of optimal and near-optimal test spaces

Assuming that we start from a well-posed and well-conditioned infinite-dimensional variational formulation (2), we look for a formulation at the finite-dimensional level which inherits these desirable features. Let $H > 0$ be a parameter (H will later be associated to the size of a mesh Ω_H of Ω). For any given finite-dimensional trial space $\mathbb{U}_H$ of dimension $\mathcal{N}$ (that depends on H), there exists a so-called optimal test space $\mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)$ of the same dimension. It is called optimal because the finite-dimensional version of problem (2),

$$\begin{aligned} &\text{Find } u_H \in \mathbb{U}_H \text{ s. t.} \\ &b(u_H, v) = f(v), \quad \forall v \in \mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H), \end{aligned} \tag{4}$$

is well posed and

$$\inf_{w_H \in \mathbb{U}_H} \sup_{v \in \mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)} \frac{b(w_H, v)}{\|w_H\|_{\mathbb{U}} \|v\|_{\mathbb{V}}} \geq \gamma.$$

In other words, the discrete inf-sup condition is bounded with the same constant γ that is involved in the infinite-dimensional problem. This implies that the discrete problem has the same stability properties as the infinite-dimensional problem. Therefore the residual $\|f - \mathcal{B}u_H\|_{\mathbb{V}'}$ is equivalent to the actual error $\|u - u_H\|_{\mathbb{U}}$ with the same constants exhibited in (3). Since these constants do not depend on H, $\|f - \mathcal{B}u_H\|_{\mathbb{V}'}$ is a *robust* error bound that is suitable for adaptivity since we can decrease H without degrading the constants of equivalence.

Unfortunately, the optimal test space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ is not computable in practice. Indeed, if $\{u_H^i\}_{i=1}^{\mathcal{N}}$ spans a basis of $\mathbb{U}_H$, then the set of functions $\{v^i\}_{i=1}^{\mathcal{N}}$ defined through the variational problems,

$$i \in \{1, \dots, \mathcal{N}\}, \quad \langle v^i, v \rangle_{\mathbb{V}} = b(u_H^i, v), \quad \forall v \in \mathbb{V} \tag{5}$$

spans a basis of $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$. Since these problems are formulated in the infinite-dimensional space $\mathbb{V}$, they cannot be computed exactly (in addition, the problems are global). To address this issue, problems (5) are $\mathbb{V}$-projected to a finite-dimensional subspace $\mathbb{V}_h$ that will be called test-search space. Therefore, in practice, an approximation $\{\bar{v}^i\}_{i=1}^{\mathcal{N}}$ to the set of functions $\{v^i\}_{i=1}^{\mathcal{N}}$ is computed by solving for all $i \in \{1, \dots, \mathcal{N}\}$,

$$\langle \bar{v}^i, v_h \rangle_{\mathbb{V}} = b(u_H^i, v_h), \quad \forall v_h \in \mathbb{V}_h. \tag{6}$$

This defines a projected test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h) := \text{span}\{\bar{v}^i\}_{i=1}^{\mathcal{N}}$. For the elliptic case and some classes of transport problems, it has been shown that it is possible to exhibit test-search spaces $\mathbb{V}_h$ (which depend on the initial $\mathbb{U}_H$) such that $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is close enough to the optimal $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ to allow that the discrete inf-sup constant

$$\gamma_H := \inf_{u_H \in \mathbb{U}_H} \sup_{\bar{v} \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)} \frac{b(u_H, \bar{v})}{\|u_H\|_{\mathbb{U}_H} \|\bar{v}\|_{\mathbb{V}_h}} \tag{7}$$

is bounded away from 0 uniformly in H. For this reason, $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is called a near-optimal test-space. In the case of transport problems, the recent work of Broersen et al. [2015] shows that good test-search spaces $\mathbb{V}_h$ can be found when they are defined over a refinement Ω_h of Ω_H.

The near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is the one that is computed in practice in the DUNE-DPG library. The finite-dimensional variational formulation that is eventually solved reads

$$\begin{aligned} &\text{Find } u_H \in \mathbb{U}_H \text{ s. t.} \\ &b(u_H, \bar{v}) = f(\bar{v}), \quad \forall \bar{v} \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h). \end{aligned} \tag{8}$$

It can be expressed as a linear system of the form $Ax = F$, $A \in \mathbb{R}^{\mathcal{N} \times \mathcal{N}}$, $x \in \mathbb{R}^{\mathcal{N}}$, $F \in \mathbb{R}^{\mathcal{N}}$. It can be proven that A is by construction symmetric positive definite. The assembly of the system and its solution in DUNE-DPG are explained in Section 3.1.

2.2 The concept of localization

Depending on the choice of $\mathbb{V}$ and $\mathbb{V}_h$, the solution of (6) to derive the near-optimal basis functions of $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ might be costly. This is because these $\mathcal{N}$ problems are, in general, global in the whole domain Ω and they cannot be decomposed into local ones. Furthermore, if the resulting near-optimal basis functions have global support, the solution of the finite-dimensional variational problem (8) is costly as well because the resulting system matrix A is full.

To prevent this, we need an appropriate variational formulation with a well-chosen test space $\mathbb{V}$ which has a product structure on the coarse grid Ω_H,

$$\mathbb{V} := \prod_{K \in \Omega_H} \mathbb{V}_K, \tag{9}$$

where $\text{supp}(v) \subset K$ for any $v \in \mathbb{V}_K$. In particular, the restriction of the $\mathbb{V}$-scalar product to $K \in \Omega_H$ has to be a scalar product for $\mathbb{V}_K$:

$$\langle \cdot, \cdot \rangle_{\mathbb{V}}|_K = \langle \cdot, \cdot \rangle_{\mathbb{V}_K}$$

The test-search space $\mathbb{V}_h$ will be chosen in such a way, that it has the same product structure as $\mathbb{V}$,

$$\mathbb{V}_h := \prod_{K \in \Omega_H} \mathbb{V}_{h,K}, \quad \mathbb{V}_{h,K} \subset \mathbb{V}_K.$$

Therefore, for any $1 \le i \le \mathcal{N}$, the near-optimal test function $\bar{v}^i$ can be written as

$$\bar{v}^i = \sum_{K\in\Omega_H} \bar{v}^i_K \chi_K,$$

where χ_K is the characteristic function of cell K. Additionally, we need a decomposition of the bilinear form as a sum over mesh cells of Ω_H,

$$b\,(u,v) = \sum_{K\in\Omega_H} b_K\,(u,v)\,, \quad \forall v \in \prod_{K\in\Omega_H} \mathbb{V}_K. \tag{10}$$

Then, for every $K \in \Omega_H$, $\bar{v}^i_K$ is the solution of a local problem in K,

$$\langle \bar{v}^i_K, v_{h,K}\rangle_{\mathbb{V}_K} = b_K\left(u^i_H, v_{h,K}\right),\ \forall v_{h,K} \in \mathbb{V}_{h,K}, \tag{11}$$

where $\{u^i_H\}_{i=1}^{\mathcal{N}}$ is a basis of $\mathbb{U}_H$. Therefore, finding $\bar{v}^i$ can be decomposed into a sum of problems, each one of which is localized on a mesh cell $K \in \Omega_H$. Moreover, if the support of u^i_H is included in some cell $K \in \Omega_H$, then the support of its corresponding near-optimal test function $\bar{v}^i$ is also a subset of K (and the neighboring cells in some cases). In other words, we would have $\bar{v}^i = \bar{v}^i_K \chi_K$ or $\bar{v}^i = \sum_{K'\in \text{ neigh(K)}} \bar{v}^i_{K'} \chi_{K'}$. Hence, if the basis functions u^i_H of $\mathbb{U}_H$ have local support, the resulting system matrix A is sparse.

2.3 An example: a linear transport equation

Let $\Omega = (0,1)^2$ and β be a vector of $\mathbb{R}^2$ with norm one. For any $x \in \partial\Omega$, let $n(x)$ be its associated outer normal vector. Then

$$\Gamma_- := \{x \in \partial\Omega \mid \beta\cdot n(x) < 0\} \subset \partial\Omega \tag{12}$$

is the inflow-boundary for the given constant transport direction β. Given $c \in \mathbb{R}$ and a function $f\colon \Omega \to \mathbb{R}$, we consider the problem of finding the solution $\varphi\colon \Omega \to \mathbb{R}$ to the simple transport equation

$$\begin{aligned} \beta\cdot\nabla\varphi + c\varphi &= f, \quad \text{in } \Omega,\\ \varphi &= 0, \quad \text{on } \Gamma_-. \end{aligned} \tag{13}$$

If we apply the DPG approach introduced in Broersen et al. [2015] to solve this problem, we first need to introduce the following spaces. Denoting by ∇_H the piecewise gradient operator, let

$$H(\beta,\Omega_H) := \{v \in L_2(\Omega) \mid \beta\cdot\nabla_H v \in L_2(\Omega)\},$$

equipped with squared "broken" norm $\|v\|^2_{H(\beta,\Omega_H)} = \|v\|^2_{L_2(\Omega)} + \|\beta\cdot\nabla_H v\|^2_{L_2(\Omega)}$. Let also

$$H_{0,\Gamma_-}(\beta,\Omega) := \text{clos}_{H(\beta,\Omega)}\{u \in H(\beta,\Omega)\cap C(\bar{\Omega}) \mid u = 0 \text{ on } \Gamma_-\}$$

and

$$H_{0,\Gamma_-}(\beta,\partial\Omega_H) := \{w|_{\partial\Omega_H} \mid w \in H_{0,\Gamma_-}(\beta,\Omega)\}$$

equipped with quotient norm

$$\|\theta\|_{H_{0,\Gamma_-}(\beta,\partial\Omega_H)} := \inf\{\|w\|_{H(\beta,\Omega)} \mid \theta = w|_{\partial\Omega_H},\ w \in H_{0,\Gamma_-}(\beta,\Omega)\}.$$

The variational formulation reads

$$\begin{aligned} &\text{For } \mathbb{U} := L^2(\Omega)\times H_{0,\Gamma_-}(\beta,\partial\Omega_H) \text{ and } \mathbb{V} := H(\beta,\Omega_H),\\ &\text{given } f \in H(\beta,\Omega_H)', \text{ find } u := (\varphi,\theta) \in \mathbb{U} \text{ such that}\\ &b(u,v) = f(v), \quad \forall v \in \mathbb{V}. \end{aligned} \tag{14}$$

In this formulation (usually called *ultra-weak* formulation) the bilinear form $b(u,v)$ is defined by

$$b(u,v) = b\left((\varphi,\theta),v\right) = \int_\Omega (-\beta\cdot\nabla v\varphi + cv\varphi)\ \mathrm{d}x + \int_{\partial\Omega_H} [\![v\beta]\!]\theta\ \mathrm{d}s. \tag{15}$$

Note that this variational formulation depends on the mesh Ω_H. Also, note the presence of an additional unknown θ that lives on the skeleton $\partial\Omega_H$ of the mesh. For smooth solutions, θ agrees with the traces of φ on $\partial\Omega_H$ (i. e. the union of cell interfaces of Ω_H).

For the discretization, we replace θ by a lifting $w \in H_{0,\Gamma_-}(\beta,\Omega)$ (for details see Broersen et al. [2015]) and take for some $m \in \mathbb{N}$,

$$\mathbb{U}_H := \left(\prod_{K\in\Omega_H} \mathbb{P}_{m-1}(K)\right) \times \left(H_{0,\Gamma_-}(\beta,\Omega) \cap \prod_{K\in\Omega_H} \mathbb{P}_m(K)\right)\Big|_{\partial\Omega_h}, \tag{16}$$

where $\mathbb{P}_m(K)$ is the space of polynomials of degree m. A viable test-search space can be taken simply as discontinuous piecewise polynomials of slightly higher degree on the finer mesh Ω_h of Ω_H, namely

$$\mathbb{V}_h := \prod_{K\in\Omega_h} \mathbb{P}_{m+1}(K). \tag{17}$$

As a result, the discrete version of (14) reads

$$\begin{aligned} &\text{Find } u_H := (\varphi_H, w_H) \in \mathbb{U}_H \text{ such that} \\ &\tilde{b}(u_H,\bar{v}) = f(\bar{v}), \quad \forall \bar{v} \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H,\mathbb{V}_h). \end{aligned} \tag{18}$$

The bilinear form $\tilde{b}$ is slightly different from b. It reads

$$\tilde{b}(u,v) = \tilde{b}\left((\varphi,w),v\right) = \int_\Omega (-\beta\cdot\nabla v\varphi + cv\varphi)\ \mathrm{d}x + \int_{\partial\Omega_h} [\![v\beta]\!]w\ \mathrm{d}s. \tag{19}$$

The trace integral is over $\partial\Omega_h$ and not $\partial\Omega_H$ since $\mathbb{V}_h$ is now a broken space with respect to the finer mesh Ω_h. This is why we need a lifting w instead of θ here. Note that depending on the polynomial degree m and the refinement level of Ω_h, there might be undefined degrees of freedom of w. In this case, they have to be fixed for example by minimizing $\|w\|_{H(\beta,\Omega)}$.

3 An Overview of the Architecture of Dune-DPG

In this section we describe how the DPG method presented in Section 2 has been implemented in Dune-DPG. As already brought up, the library has been built upon the finite element package Dune.

The user starts by choosing the appropriate test-search space $\mathbb{V}_h$ and trial space $\mathbb{U}_H$ for his problem. Then, the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ are declared via the classes `BilinearForm` and `InnerProduct` (see Section 3.1.2). They both consist of an arbitrary number of elements of the type `IntegralTerm` (see Section 3.1.3). The `DPGSystemAssembler` class handles the automatic assembly of the linear system $Ax = F$ associated to problem (8). As Section 3.1.1 explains, it includes:

- the assembly of the matrix A and the right hand side F
- the treatment of boundary conditions

To assemble A, we use the decomposition (10) and define the local matrices A_K by

$$(A_K)_{i,j} = b_K(u^j_{H,K}, \bar{v}^i_K) \tag{20}$$

where $\{u^i_{H,K}\}_{i=1}^{N_K}$ is a basis for the restriction of $\mathbb{U}_H$ to K and $\{\bar{v}^i_K\}_{i=1}^{N_K}$ is a basis for the restriction of the near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ to K. To compute these local matrices, we follow the same lines as the Camellia library [Roberts, 2014] and start from a basis $\{v^j_{h,K}\}_{j=1}^{\mathcal{M}_K}$ of the test-search space $\mathbb{V}_h$ restricted to K. Then, we use the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ to construct the matrices B_K and G_K defined by

$$(B_K)_{i,j} = b_K(u^j_{H,K}, v^i_{h,K}), \qquad (G_K)_{i,j} = \langle v^i_{h,K}, v^j_{h,K}\rangle_{\mathbb{V}}. \tag{21}$$

It follows from (11) that the coefficients $c^{i,j}_K$ of the basis functions $\bar{v}^i_K = \sum_{j=1}^{\mathcal{M}_K} c^{i,j}_K v^j_{h,K}$ of the near-optimal test space restricted to K are the columns of the matrix

$$C_K := G_K^{-1} B_K \tag{22}$$

and the local matrices satisfy

$$A_K = A_K^T = B_K^T C_K. \tag{23}$$

The matrices C_K and A_K are computed by the class `TestspaceCoefficientMatrix` which also offers a feature to store those matrices to reduce computational costs in case of constant coefficients (see Section 3.1.4). Finally, the `DPGSystemAssembler` class constructs the global matrix A out of the local matrices A_K. In addition to these classes, the class `ErrorTools` handles the computation of a posteriori estimators following the guidelines that are given in Section 3.2.

3.1 Assembling the discrete system for a given PDE

The following subsections describe the `DPGSystemAssembler` class and all the classes used by it.

3.1.1 DPGSystemAssembler The assembly of the discrete system $Ax = F$ derived from the variational problem is handled by the class `DPGSystemAssembler`. Before we can create a `DPGSystemAssembler` by calling the method `make_DPGSystemAssembler` we first have to define the trial space $\mathbb{U}_H$, the near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ as well as the bilinear form b and the inner product that describe our DPG system. The bilinear form is an object of the class `BilinearForm` which is explained in Section 3.1.2. As for the spaces $\mathbb{U}_H$ and $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$, they are both given by a `std::tuple` composed of (scalar) global basis functions from DUNE-FUNCTIONS. The reason to use a tuple is to handle problems involving several unknowns. For instance, in the ultra-weak formulation introduced for the transport problem in Section 2.3, we have two unknowns (φ, w) and $\mathbb{U}_H$ is a product of two spaces.

The following lines of code taken from `src/plot_solution.cc` give an overview of how to set up the transport problem from Section 2.3. The individual steps will be explained in the following subsections. The example uses UG (see Bastian et al. [1997]) to represent the mesh but the code admits other grid implementations. We refer to the `README` file included in the library for details on this and on how to launch the program.

src/plot_solution.cc

```
128   using FEBasisInterior = Functions::LagrangeDGBasis<GridView, 1>;
129   FEBasisInterior spacePhi(gridView);
130
131   using FEBasisTraceLifting = Functions::PQkNodalBasis<GridView, 2>;
132   FEBasisTraceLifting spaceW(gridView);
133
134   auto solutionSpaces = std::make_tuple(spacePhi, spaceW);
135
```

```
136   using FEBasisTest
137       = Functions::PQkDGRefinedDGBasis<GridView, 1, 3>;
138   auto testSearchSpaces = std::make_tuple(FEBasisTest(gridView));
139
140   auto bilinearForm = make_BilinearForm(testSearchSpaces, solutionSpaces,
141           make_tuple(
142               make_IntegralTerm<0,0,IntegrationType::valueValue,
143                                 DomainOfIntegration::interior>(c),
144               make_IntegralTerm<0,0,IntegrationType::gradValue,
145                                 DomainOfIntegration::interior>(-1., beta),
146               make_IntegralTerm<0,1,IntegrationType::normalVector,
147                                 DomainOfIntegration::face>(1., beta)));
148   auto innerProduct = make_InnerProduct(testSearchSpaces,
149           make_tuple(
150               make_IntegralTerm<0,0,IntegrationType::valueValue,
151                                 DomainOfIntegration::interior>(1.),
152               make_IntegralTerm<0,0,IntegrationType::gradGrad,
153                                 DomainOfIntegration::interior>(1., beta)));
154
155   typedef GeometryBuffer<GridView::template Codim<0>::Geometry> GeometryBuffer;
156   GeometryBuffer geometryBuffer;
157
158   auto systemAssembler
159       = make_DPGSystemAssembler(bilinearForm, innerProduct, geometryBuffer);
```

The `systemAssembler` has a member variable `testspaceCoefficientMatrix_` which is of type `TestspaceCoefficientMatrixBuffered` or `TestspaceCoefficientMatrixUnbuffered`, depending on whether `make_DPGSystemAssembler` is called with or without a `GeometryBuffer`, for details on that, see Section 3.1.4.

Once `systemAssembler` has been defined, a call to the method `assembleSystem(stiffnessMatrix, rhsVector, rhsFunctions)` assembles the matrix A and the right-hand side vector F. They are stored in the variables `stiffnessMatrix` (of type `BCRSMatrix<FieldMatrix<double,1,1>>`) and `rhsVector` (of type `BlockVector<FieldVector<double,1>>`). The input parameter `rhsFunctions` is of type `LinearForm` and represents the function f from the right hand side of the PDE. Internally, the class `DPGSystemAssembler` iterates over all mesh cells K and delegates the work of computing local contributions A_K to the system matrix A to `testspaceCoefficientMatrix_.systemMatrix()`. Similarly the local right-hand side vectors are computed by taking the product of the precomputed `testspaceCoefficientMatrix_.coefficientMatrix()` with `LinearForm::getLocalVector()`. For constructing A and F out of the local matrices A_K and the local right-hand side vectors, we make use of the mapping between local and global degrees of freedom given by the index sets from DUNE-FUNCTIONS.

`DPGSystemAssembler` is also responsible for applying boundary conditions to the system. So far, only Dirichlet boundary conditions are implemented. To this end, first the degrees of freedom affected by the boundary condition are marked. Then the boundary values are set to the corresponding nodes with the method `applyDirichletBoundary`. For instance, if we are considering the transport problem of Section 2.3, we need to set to zero the degrees of freedom of w that are in Γ_-. For this, we mark the relevant nodes with the method `getInflowBoundaryMask` and store the information in a vector `dirichletNodesInflow`. Then we call `applyDirichletBoundary` as we outline in the following listing. The sequence of instructions is given in the code below. Note that the trial space associated to w is required. Since, in our ordering, w is our second unknown, we get its associated trial space with the command `std::get<1>(solutionSpaces)` (since `std::tuple` starts counting from 0).

src/plot_solution.cc

```
165   typedef BlockVector<FieldVector<double,1> > VectorType;
166   typedef BCRSMatrix<FieldMatrix<double,1,1> > MatrixType;
167
168   VectorType rhsVector;
169   MatrixType stiffnessMatrix;
```

```
170
171 auto rhsFunctions
172   = make_DPGLinearForm(testSearchSpaces,
173         std::make_tuple(make_LinearIntegralTerm<0,
174                           LinearIntegrationType::valueFunction,
175                           DomainOfIntegration::interior>(f(beta))));
176 systemAssembler.assembleSystem(stiffnessMatrix, rhsVector, rhsFunctions);
177
178 // Determine Dirichlet dofs for w (inflow boundary)
179 {
180   std::vector<bool> dirichletNodesInflow;
181   BoundaryTools boundaryTools = BoundaryTools();
182   boundaryTools.getInflowBoundaryMask(std::get<1>(solutionSpaces),
183                                       dirichletNodesInflow,
184                                       beta);
185   systemAssembler.applyDirichletBoundary<1>
186       (stiffnessMatrix,
187        rhsVector,
188        dirichletNodesInflow,
189        0.);
190 }
```

Finally, in certain types of problems, some degrees of freedom might be ill-posed. For example, in the transport case, the degrees of freedom corresponding to trial functions on faces aligned with the flow direction will be weighted with 0 coefficients in the matrix or interior degrees of freedom of the lifting w of the trace variable may be undefined. To address these issues, `DPGSystemAssembler` provides several methods like `defineCharacteristicFaces` to interpolate undefined degrees of freedom on characteristic faces or `applyMinimization` to handle undefined degrees of freedom in the interior and optionally also on characteristic faces by minimizing a given norm.

Once A and F are obtained, the system $Ax = F$ (which is, from the theory, invertible) can be solved with the user's favorite direct or iterative scheme.

3.1.2 BilinearForm and InnerProduct

As it follows from (10), the bilinear form $b(\cdot,\cdot)$ can be decomposed into local bilinear forms $b_K(\cdot,\cdot)$. The `BilinearForm` class describes b_K and provides access to the corresponding local matrices A_K defined in (20) which are then used by the `DPGSystemAssembler` to assemble the global matrix A.

In our case, we view a bilinear form b_K as a sum of what we will call elementary integral terms. By this we mean integrals over K (or ∂K) which are a product of a test search function $v \in \mathbb{V}_h$ (or its derivatives) and a trial function $u \in \mathbb{U}_H$ (or its derivatives). Additionally, the product might also involve some given function c. The current release only supports constant functions c and the non-constant case will be delivered in a future release. For instance, in our transport equation (cf. (19)),

$$b_K(u,v) = b_K((\varphi,w),v) := \underbrace{\int_K cv\varphi}_{Int_0} - \underbrace{\int_K \beta\cdot\nabla v\varphi}_{Int_1} + \underbrace{\sum_{K_h\in\Omega_h, K_h\subset K}\int_{\partial K_h} vw\beta\cdot n}_{Int_2}, \tag{24}$$

where we have omitted the tilde to ease notation here. Therefore the matrix $A_K = \sum_{i\in I} A_K^i$ can be computed as a sum of the matrices A_K^i corresponding to the different elementary integrals Int_i, $i \in I$. Any of the elementary integrals can be expressed via the class `IntegralTerm` that we describe in Section 3.1.3.

To create an object `bilinearForm` of the class `BilinearForm`, we call `make_BilinearForm` as follows.

C++ code

```
1 auto bilinearForm = make_BilinearForm (testSearchSpaces, solutionSpaces, terms);
```

The variables `testSearchSpaces` and `solutionSpaces` are the ones introduced in Section 3.1.1 to represent $\mathbb{V}_h$ and $\mathbb{U}_H$. The object `terms` is a tuple of objects of the class `IntegralTerm`. Once that the object `bilinearForm` exists, a call to the method `getLocalMatrix` computes A_K by iterating over all elementary integral terms and summing up their contributions A_K^i.

Let us now briefly discuss the class `InnerProduct`. Its aim is to allow the computation of the inner products associated to the Hilbert spaces $\mathbb{U}$ and $\mathbb{V}$. For this, we take advantage of the fact that an inner product can be seen as a symmetric bilinear form $b(u, v)$ where u and v are both functions from some space. Hence, we can reuse the structure of `BilinearForm` for summing over elementary integral terms to define the class `InnerProduct`. The construction of an `InnerProduct` is thus done with

C++ code

```cpp
auto innerProduct = make_InnerProduct (testSpaces, terms);
```

3.1.3 IntegralTerm An `IntegralTerm` represents an elementary integral over the interior of a cell K, over its faces ∂K or even over faces of a partition of K. It expresses a product between a term related to a test function v and a term related to a trial function u. Examples are Int_0, Int_1 and Int_2 from (24).

The `IntegralTerm` is parametrized by two `size_t` that give the indices of the test and trial spaces that we want to integrate over. Additionally we specify the type of evaluations used in the integral with a template parameter of type

C++ code

```cpp
enum class IntegrationType {
    valueValue,
    gradValue,
    valueGrad,
    gradGrad,
    normalVector,
    normalSign
};
```

and the domain of integration with a template parameter of type

C++ code

```cpp
enum class DomainOfIntegration {
    interior,
    face
};
```

If `integrationType` is of type `IntegrationType::valueValue` or `IntegrationType::normalSign`, the function `make_IntegralTerm` has to be called as follows:

C++ code

```cpp
auto integralTerm
   = make_IntegralTerm<lhsSpaceIndex, rhsSpaceIndex,
                       integrationType, domainOfIntegration>(c);
```

where c is a scalar coefficient in front of the test space product and is of arithmetic type, e. g. `double`. The template parameter `domainOfIntegration` is one of the types from `DomainOfIntegration` and the parameters `lhsSpaceIndex` and `rhsSpaceIndex` refer, in this particular order, to the indices of test and trial space in their respective tuples of test and trial spaces. Note that the objects of the class `IntegralTerm` are not given the spaces themselves but only some indices referring to them. This is because the spaces are managed by the class `BilinearForm` (or `InnerProduct`) owning the `IntegralTerm`.

For other `integrationTypes`, we also need to specify the flow direction `beta` by calling

C++ code

```
1 auto integralTerm
2    = make_IntegralTerm<lhsSpaceIndex, rhsSpaceIndex,
3                        integrationType, domainOfIntegration>(c, beta);
```

where `c` is again of arithmetic type and `beta` is of vector type, e. g. `FieldVector<double, dim>`.

The `IntegralTerm` Int_1 from example (24) can be created with

C++ code

```
1 auto integralTerm
2    = make_IntegralTerm<0, 0, IntegrationType::gradValue,
3                        DomainOfIntegration::interior>(-1., beta);
```

where the two zeroes are, in this particular order, the indices of test and trial space in their respective tuples of test and trial spaces.

The class `IntegralTerm` provides a method `getLocalMatrix` that computes its contribution A_K^i to the local matrix A_K and that is called by the `getLocalMatrix` method of `BilinearForm` or `InnerProduct`. Since one would normally want to write a program which solves a fixed problem, we use Boost Fusion[2] to do as much work at compile time as possible. Since the `IntegrationType` and `DomainOfIntegration` of each `IntegralTerm` are compile time constants, an optimizing C++ compiler should be able to optimize away some unused code branches. Defining the bilinear form and inner product at compile time also gives us the opportunity to check for errors in the problem formulation at compile time. For now this is not done, but we would like to include such checks in the future to improve the usability of our library.

3.1.4 TestspaceCoefficientMatrix (buffered and unbuffered)

For any element K, the computation of the matrices C_K and A_K defined in (22) and (23) is handled either by the class `TestspaceCoefficientMatrixBuffered` or by `TestspaceCoefficientMatrixUnbuffered`. Two classes have been developed to minimize computations when the PDE coefficients are constant. In this case C_K and A_K depend only on the geometry of the element K. As a result, the value of C_K and A_K will be constant for all elements K having, up to a translation, the same map to the reference element. Thus, C_K and A_K can be computed only once for all cells sharing this mapping property. The class `TestspaceCoefficientMatrixBuffered` makes use of the above and reduces computational costs in grids where many cells have the same mapping property. The class `TestspaceCoefficientMatrixUnbuffered` handles the general case (grids with many different cell mapping types and, in future releases, non-constant PDE coefficients).

Structurally speaking, both classes have the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ as template parameters and offer a method `bind(const Entity& e)` in which they use the methods `BilinearForm::getLocalMatrix()` and `InnerProduct::getLocalMatrix()` to set up B_K and G_K as defined in (21). They compute $C_K = G_K^{-1} B_K$ via the Cholesky algorithm and $A_K = B_K^T C_K$. The computed matrices can be accessed via the methods `coefficientMatrix()` and `systemMatrix()`, respectively. The constructor of `TestspaceCoefficientMatrixUnbuffered` gets only the bilinear form and the inner product. In addition to this, `TestspaceCoefficientMatrixBuffered` needs a `GeometryBuffer` containing a map to save the geometry of the elements K and the corresponding matrices C_K and A_K. When the `TestspaceCoefficientMatrixBuffered` is bound to a new element K, then first it is checked whether its map to the reference element is already in the buffer. If so, the saved matrices C_K and A_K are used. Otherwise, they are computed and added to the map.

[2] Fusion is a meta programming library and part of the C++ library collection Boost: http://www.boost.org/

3.2 A posteriori error estimators

To compute the residual

$$\|f - \mathcal{B}u_H\|_{\mathbb{V}'} = \sup_{v \in \mathbb{V}} \frac{\|f(v) - b(u_H, v)\|_{\mathbb{V}}}{\|v\|_{\mathbb{V}}},$$

we exploit once again the product structure of $\mathbb{V}$ and use the fact that

$$\|f - \mathcal{B}u_H\|^2_{\mathbb{V}'} = \sum_{K \in \Omega_H} \|r_K(u_H, f)\|^2_{\mathbb{V}'_K} = \sum_{K \in \Omega_H} \|R_K(u_H, f)\|^2_{\mathbb{V}_K}$$

where r_K is the cell-wise residual. R_K is the Riesz-lift of r_K in $\mathbb{V}_K$ so it is the solution of

$$\langle R_K(u_H, f), v \rangle_{\mathbb{V}_K} = b(u_H, v) - f(v), \quad \forall v \in \mathbb{V}_K. \tag{25}$$

Since (25) is an infinite-dimensional problem, we project the Riesz-lift R_K to a finite-dimensional subspace $\overline{\mathbb{V}}_K$ of $\mathbb{V}_K$, obtaining an approximation $\overline{R}_K$. This in turn gives the a posteriori error estimator

$$\|\overline{R}(u_H, f)\|_{\mathbb{V}} := \left(\sum_{K \in \Omega_H} \|\overline{R}_K(u_H, f)\|^2_{\mathbb{V}_K} \right)^{1/2}. \tag{26}$$

An appropriate choice of the a posteriori search space $\overline{\mathbb{V}}_K$ depends on the problem and is crucial to make $\|\overline{R}_K\|_{\mathbb{V}_K}$ good error indicators.

In DUNE-DPG, the computation of the a posteriori estimator (26) is handled by the class `ErrorTools`. The following lines of code compute (26) for the solution `u_H` of a problem with bilinear form `bilinearForm`, inner product `innerProduct` and right hand side `rhsVector`.

C++ code

```
1 ErrorTools errorTools = ErrorTools();
2 double aposterioriErr =
      errorTools.aPosterioriError(bilinearForm,innerProduct,u_H,rhsVector);
```

The object `bilinearForm` is of the type `BilinearForm` described above. It has to be created with an object `testSpace` associated to the a posteriori search space $\overline{\mathbb{V}}_H$. The same applies for `innerProduct`, which is of type `InnerProduct`. Also the vector `rhsVector` has to be set up using the a posteriori search space.

Furthermore, `ErrorTools` contains a method `DoerflerMarking` to use the approximate cell-wise residuals $\|\overline{R}_K\|_{\mathbb{V}_K}$ for adaptive refinement.

4 Numerical Example: Implementation of Pure Transport in DUNE-DPG

As a simple numerical example, we solve the transport problem (13) with

$$\begin{aligned} c &= 0, \\ \beta &= (\beta_1, \beta_2) = (\cos(\pi/8), \sin(\pi/8)), \\ f &= 1. \end{aligned}$$

As Figure 1d shows, the exact solution

$$\varphi(x) = \begin{cases} \sqrt{\beta_2^2/\beta_1^2 + 1} \cdot x_1, & \text{if } \beta_1 \cdot x_2 - \beta_2 \cdot x_1 > 0 \\ \sqrt{\beta_1^2/\beta_2^2 + 1} \cdot x_2, & \text{else} \end{cases}$$

describes a linear ramp starting at 0 in each point of the inflow boundary Γ_- and increasing with slope 1 along the flow direction β. There is a kink in the solution starting in the lower left corner of Ω and propagating along β.

For the numerical solution, we let Ω_H be a partition of Ω into uniformly shape regular triangles generated by partitioning $\Omega = [0,1]^2$ into $n \times n$ uniform quadrangles that are then each divided from the lower left to the upper right corner into two triangles. Ω_h is a refinement of Ω_H to some level $\ell \in \mathbb{N}_0$ such that $h = 2^{-\ell}H$, i. e. each triangle in Ω_H gets subdivided into 4^ℓ congruent subtriangles.

With $\mathbb{U}_H$ and $\mathbb{V}_h$ defined as in (16) and (17) with $m = 2$, we compute $u_H = (\varphi_H, w_H) \in \mathbb{U}_H$ by solving the ultra-weak variational formulation (18). We investigate convergence in H of the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$. We also evaluate the a posteriori estimator $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ when the components $\overline{R}_K(u_H, f)$ are computed with a subspace $\overline{\mathbb{V}}_K$ of polynomials of degree 5, $\forall K \in \Omega_H$.

Regarding the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$, as Figure 1a shows, we observe linear convergence as H decreases. This is to be expected since the polynomial degree to compute φ_H is 1. The figure also shows that the refinement level ℓ of the test-search space $\mathbb{V}_h$ has essentially no impact on the behavior of the error.

Regarding the behavior of the a posteriori estimator $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$, it is possible to see in Figures 1b and 1c that the quality of $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ slightly degrades as H decreases in the sense that, as H decreases, $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ represents the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$ less and less faithfully. An element that might be playing a role is that $\|\overline{R}_K(u_H, f)\|_{\mathbb{V}}$ is not exactly an estimation of $\|\varphi - \varphi_H\|_{L_2(\Omega)}$, but of the error including also w_H, namely $\|u - u_H\|_{\mathbb{U}} = \|(\varphi, w) - (\varphi_H, w_H)\|_{\mathbb{U}}$.

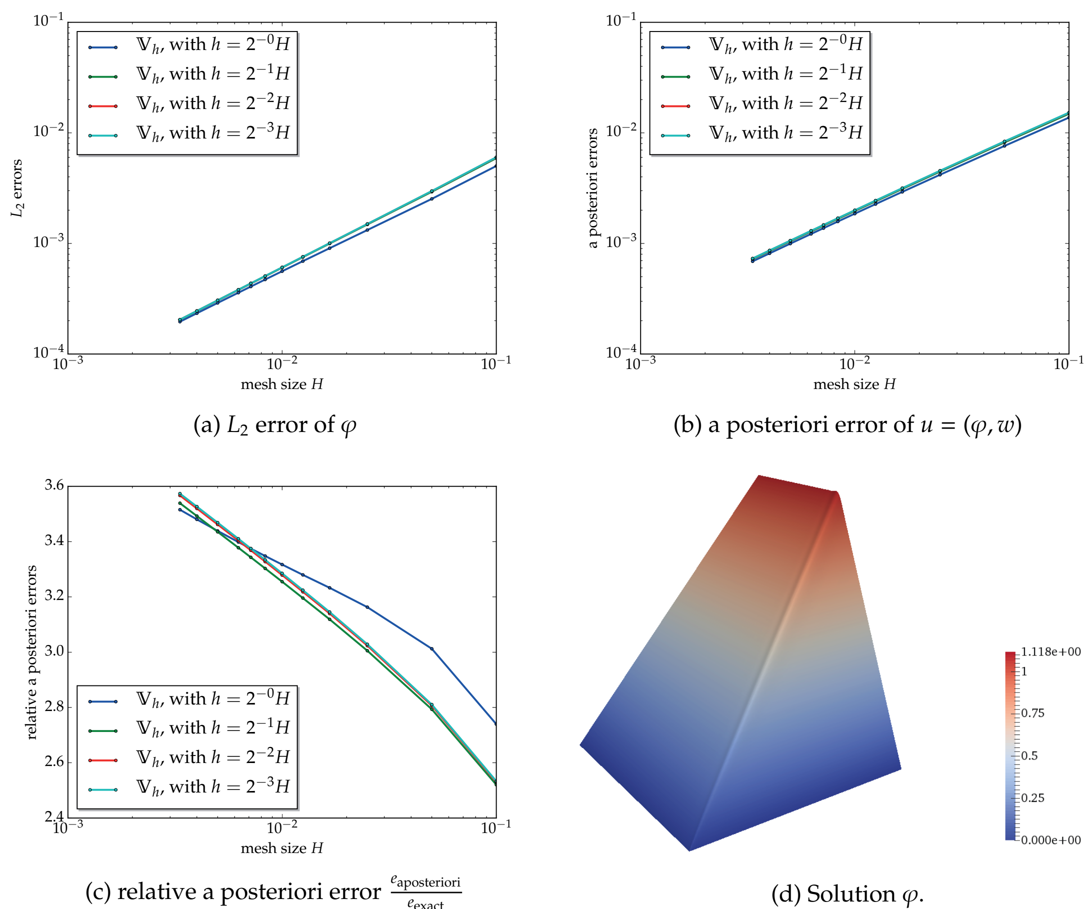
The source code used to compute the values from the convergence plots in Figure 1a–c can be found in `src/convergence_test.cc` while the code used to generate Figure 1d can be found in `src/plot_solution.cc`. In this program, the user can easily experiment with the transport equation by changing the values of β and c (see the `README` file).

In Figure 2 we compare the runtime for assembling the system with and without buffering of the test space coefficient matrices from Section 3.1.4 on a 4 core Intel Core i7-3770 CPU with 3.40GHz and 15GB of RAM. These performance measurements have been done for the same problem as before and the corresponding code resides in `src/profile_testspacecoefficientmatrix.cc`. The plots show the clear advantage of using the buffering whenever the grid is uniform and we use constant coefficients.

5 Conclusion And Future Work

In Section 2, we gave a short overview of the DPG method. We then introduced our Dune-DPG library in Section 3, documenting the internal structure and showing how to use it to solve a given PDE. Finally, we showed some numerical convergence results computed for a problem with well-known solution. This allowed us to compare our a posteriori estimators to the real L_2 error of our numerical solution. As a next step, we want to fully implement non-constant coefficients in order to support a wider range of PDEs.

Furthermore, we want to improve our handling of vector valued problems with one notable example being first order formulations of convection–diffusion problems. With our current `std::tuple` of spaces structure used throughout the code, we have to implement vector valued spaces by adding the same scalar valued space several times. With the Dune-TypeTree library from Müthing [2015] we can handle vector valued spaces much more easily, as has already been shown in Dune-Functions. This will result in major changes in our code, but will probably allow us to replace our dependency on Boost Fusion with more modern C++11 constructs. In the long run, we hope that this would give us increased maintainability and decreased compile times in addition to the improvements in the usability of vector valued problems. Finally, we want to explore parallelization to increase the performance. This is aligned with our long-term goal of



(a) L_2 error of φ

(b) a posteriori error of $u = (\varphi, w)$

(c) relative a posteriori error $\frac{e_{\text{aposteriori}}}{e_{\text{exact}}}$

(d) Solution φ.

Figure 1: L_2 error and a posteriori error estimator of numerical solutions

making DUNE-DPG a flexible building block for constructing DPG solvers for a large range of different problem types.

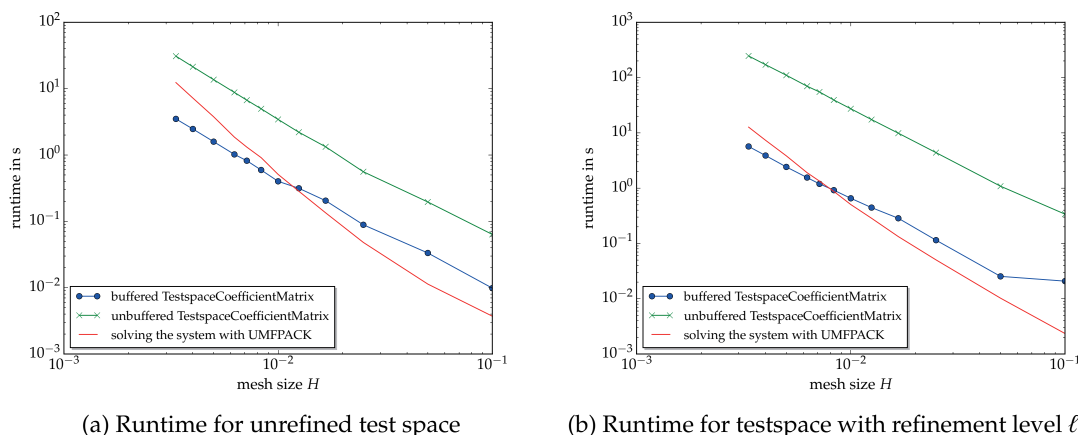
Acknowledgments

We thank O. Sander for his introduction to the DUNE library and his guidance in understanding it. We also thank W. Dahmen for introducing us to the topic of DPG and our anonymous reviewers for their constructive comments which helped to significantly improve the paper and the efficiency of the computation of test functions in our code. Finally, O. Mula is indebted to the AICES institute of RWTH Aachen for hosting her as a postdoc during 2014–2015 which is the period in which large parts of DUNE-DPG were developed.

References

P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1(1):27–40, 1997. doi: 10.1007/s007910050003.

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, 2008. doi: 10.1007/s00607-008-0004-9.



(a) Runtime for unrefined test space

(b) Runtime for testspace with refinement level $\ell = 2$

Figure 2: Runtime for assembling the system with buffered and unbuffered testspace coefficient matrices.

M. Blatt and P. Bastian. The iterative solver template library. In *International Workshop on Applied Parallel Computing*, pages 666–675. Springer, 2006. doi: 10.1007/978-3-540-75755-9_82.

M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. The Distributed and Unified Numerics Environment, version 2.4. *Archive of Numerical Software*, 4(100):13–29, 2016. doi: 10.11588/ans.2016.100.26526.

D. Broersen and R. Stevenson. A robust Petrov–Galerkin discretisation of convection–diffusion equations. *Computers & Mathematics with Applications*, 68(11):1605–1618, 2014. doi: 10.1016/j.camwa.2014.06.019.

D. Broersen and R. P. Stevenson. A Petrov–Galerkin discretization with optimal test space of a mild-weak formulation of convection-diffusion equations in mixed form. *IMA Journal of Numerical Analysis*, 35(1):39–73, 2015. doi: 10.1093/imanum/dru003.

D. Broersen, W. Dahmen, and R. P. Stevenson. On the stability of DPG formulations of transport equations. IGPM Preprint 433, IGPM, RWTH Aachen, Oct. 2015. URL https://www.igpm.rwth-aachen.de/forschung/preprints/433.

C. Carstensen, L. Demkowicz, and J. Gopalakrishnan. A posteriori error control for DPG methods. *SIAM Journal on Numerical Analysis*, 52(3):1335–1353, 5 June 2014. doi: 10.1137/130924913.

C. Carstensen, L. Demkowicz, and J. Gopalakrishnan. Breaking spaces and forms for the DPG method and applications including Maxwell equations. *Computers & Mathematics with Applications*, 72(3):494–522, 2016. doi: 10.1016/j.camwa.2016.05.004.

W. Dahmen, C. Huang, C. Schwab, and G. Welper. Adaptive Petrov–Galerkin methods for first order transport equations. *SIAM Journal on Numerical Analysis*, 50(5):2420–2445, 2012. doi: 10.1137/110823158.

W. Dahmen, C. Plesken, and G. Welper. Double greedy algorithms: Reduced basis methods for transport dominated problems. *ESAIM: Mathematical Modelling and Numerical Analysis*, 48(3):623–663, 2014. doi: 10.1051/m2an/2013103. URL http://www.igpm.rwth-aachen.de/forschung/preprints/357.

L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov–Galerkin methods. Part II: Optimal test functions. *Numerical Methods for Partial Differential Equations*, 27(1):70–105, Jan. 2011. doi: 10.1002/num.20640.

L. Demkowicz and J. Gopalakrishnan. Discontinuous Petrov–Galerkin (DPG) method. ICES Report 15-20, ICES, UT Austin, Oct. 2015. URL https://www.ices.utexas.edu/media/reports/2015/1520.pdf.

L. Demkowicz, J. Gopalakrishnan, I. Muga, and J. Zitelli. Wavenumber explicit analysis of a DPG method for the multidimensional Helmholtz equation. *Computer Methods in Applied Mechanics and Engineering*, 213–216:126–138, Mar. 2012. doi: 10.1016/j.cma.2011.11.024.

C. Engwer, C. Gräser, S. Müthing, and O. Sander. The interface for functions in the dune-functions module. Dec. 2015. URL http://arxiv.org/abs/1512.06136.

J. Gopalakrishnan and W. Qiu. An analysis of the practical DPG method. *Mathematics of Computation*, 83(286):537–552, 2014. doi: 10.1090/S0025-5718-2013-02721-4. URL http://arxiv.org/abs/1107.4293.

S. Müthing. *A Flexible Framework for Multi Physics and Multi Domain PDE Simulations*. PhD thesis, Universität Stuttgart, Feb. 2015.

A. H. Niemi, N. O. Collier, and V. M. Calo. Automatically stable discontinuous Petrov–Galerkin methods for stationary transport problems: Quasi-optimal test space norm. *Computers & Mathematics with Applications*, 66(10):2096–2113, 2013. doi: 10.1016/j.camwa.2013.07.016.

N. V. Roberts. Camellia: A software framework for discontinuous Petrov–Galerkin methods. *Computers & Mathematics with Applications*, 68(11):1581–1604, 2014. doi: 10.1016/j.camwa.2014.08.010.

N. V. Roberts, T. Bui-Thanh, and L. Demkowicz. The DPG method for the Stokes problem. *Computers & Mathematics with Applications*, 67(4):966–995, 2014. doi: 10.1016/j.camwa.2013.12.015.

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 129–149
DOI: 10.11588/ans.2017.1.28068

Using Dune-Fem for Adaptive Higher Order Discontinuous Galerkin Methods for Two-phase Flow in Porous Media

Birane Kane[1]

[1]Institute of Applied Analysis and Numerical Simulation, University of Stuttgart

Received: February 17th, 2016; **final revision:** November 10th, 2016; **published:** March 6th, 2017.

Abstract: In this document we present higher order Discontinuous Galerkin discretization of a two-phase flow model describing subsurface flow in strongly heterogeneous porous media. The flow in the domain is immiscible and incompressible with no mass transfer between phases. We consider a fully implicit, locally conservative, higher order discretization on adaptively generated meshes. The implementation is based on the open-source PDE software framework Dune.

1 Introduction

Simulation of multi-phase flows and transport processes in porous media requires careful numerical treatment due to the strong heterogeneity of the underlying porous medium. The spatial discretization requires locally conservative methods in order to be able to follow small concentrations [6]. Discontinuous Galerkin (DG), Finite Volume and Mixed Finite Element, are examples of discretization methods which achieve local conservation at the element level [13]. The first DG method was originally developed for solving the neutron transport problem [19]. Since then, numerous DG methods have been developed for hyperbolic problems, the Bassi and Rebay [5] method and the Local Discontinuous Galerkin (LDG) method introduced in [11] are some examples among others. Independently of the development of the DG methods for hyperbolic equations, Interior Penalty (IP) Discontinuous Galerkin methods for elliptic and parabolic equations were introduced in [2], [4], [14], [23]. DG methods present attractive features such as an inherent local and global conservation, a high-order accuracy, a high parallel efficiency and a geometric flexibility (unstructured meshes and non-conforming grids) allowing an easier local *hp*-adaptivity. Furthermore, the ability of DG methods to treat rough coefficient problems and capture discontinuities in solutions allows them to be suitable candidates for the discretization of PDE's arising in Environmental Engineering.

Application of DG methods to incompressible two-phase flow started with [7], [18], [21]. The initial approach consisted in a decoupled formulation where first a pressure equation is solved implicitly and then the saturation is advanced by an explicit time stepping scheme (Implicit

Pressure Explicit Saturation). Upwinding, slope limiting techniques, and sometimes H(div) projection were required in order to remove unphysical oscillations and to ensure convergence. More recently, Bastian [8] presented a fully coupled symmetric interior penalty DG formulation for incompressible two-phase flow based on a formulation using a wetting-phase potential/capillary potential formulation. Discontinuity in capillary pressure functions is taken into account by incorporating the interface conditions into the penalty terms for the capillary potential. Heterogeneity in absolute permeability is treated by weighted averages. A higher-order diagonally implicit Runge-Kutta method in time is used and there is no post processing of the velocity or slope limiting. The author did not use any kind of adaptivity and only piecewise linear and piecewise quadratic functions are employed.

In this work, we implement and evaluate numerically Interior Penalty DG methods for 2d and 3d incompressible, immiscible, two-phase flow. We consider a strongly heterogeneous porous medium and discontinuous capillary pressure functions. We write the system in terms of a phase-pressure/phase-saturation formulation. Adams-Moulton schemes of first and second order in time are combined with various Interior Penalty DG discretizations in space such as the Symmetric Interior Penalty Galerkin (SIPG), the Nonsymmetric Interior Penalty Galerkin (NIPG) and the Incomplete Interior Penalty Galerkin (IIPG) [3]. This implicit space time discretization leads to a fully coupled nonlinear system requiring to build a Jacobian matrix at each time step for the Newton-Raphson method. We include in our implementation local mesh adaptivity on non-conforming grids. To our knowledge, this is the first time the concept of local h-adaptivity is incorporated in the DG discretization of a 3d two-phase flow with strong heterogeneity, discontinuous capillary pressure functions and gravity effects. The milestone contribution of Klieber & Rivière [18] restrained itself to a decoupled formulation with continuous capillary pressure functions and only 2d flow on non-conforming simplicials grids were considered. We use higher order polynomial degree up to piecewise cubics.

The rest of this document is organised as follows. In the next section, we describe the two-phase flow model. The DG discretization is introduced in section 3. The adaptive strategy in space is outlined in section 4. The implementation with `Dune-Fem` is described in section 5. Numerical examples are provided in section 6. Finally concluding remarks are provided in the last section.

2 Problem Setting

This section introduces the mathematical formulation of a two-phase Darcy problem modeling porous-media flow. The flow is immiscible and incompressible with no mass transfer between phases.

2.1 Two-phase flow model

We consider an open and bounded domain $\Omega \in \mathbb{R}^d$, $d \in \{1,2,3\}$ and the time interval $\mathcal{J} = (0,T)$, $T > 0$. The flow of the wetting-phase and the nonwetting-phase is described by the Darcy's law and the continuity equation for each phase, namely,

$$v_\alpha = -\lambda_\alpha K(\nabla p_\alpha - \rho_\alpha g), \tag{2.1}$$

$$\phi \frac{\partial \rho_\alpha s_\alpha}{\partial t} + \nabla \cdot (\rho_\alpha v_\alpha) = \rho_\alpha q_\alpha, \tag{2.2}$$

$$\sum_\alpha s_\alpha = 1, \tag{2.3}$$

$$p_n - p_w = p_c(s_{w,e}). \tag{2.4}$$

Here, we search for the phase pressures p_α and the phase saturations s_α, $\alpha \in \{w, n\}$. We denote with subscript w the wetting-phase and with subscript n the nonwetting-phase. K is the permeability of the porous medium, ρ_α is the phase density, q_α is a source/sink term and g is the constant

 © by the authors, 2017

gravitational vector. We assume the porosity ϕ is time independent and uniformly bounded from above and below; that is there exist $\phi_1, \phi_2 > 0$ such that:

$$0 < \phi_1 \leq \phi \leq \phi_2.$$

Phase mobilities λ_α are defined by

$$\lambda_\alpha = \frac{k_{r\alpha}}{\mu_\alpha}, \alpha \in \{w, n\}, \tag{2.5}$$

where μ_α is the phase viscosity and $k_{r\alpha}$ is the relative permeability of phase α. The relative permeabilities are functions that depend nonlinearly on the phase saturation (i.e. $k_{r\alpha} = k_{r\alpha}(s_\alpha)$). Models for the relative permeability are the van-Genuchten model [22] and the Brooks-Corey model [10]. For example, in the Brooks-Corey model,

$$k_{rw}(s_{w,e}) = s_{w,e}^{\frac{2+3\theta}{\theta}}, \quad k_{rn}(s_{n,e}) = (s_{n,e})^2(1 - (1 - s_{n,e})^{\frac{2+\theta}{\theta}}), \tag{2.6}$$

where the effective saturation $s_{\alpha,e}$ is

$$s_{\alpha,e} = \frac{s_\alpha - s_{\alpha,r}}{1 - s_{w,r} - s_{n,r}}, \qquad \forall \alpha \in \{w, n\}. \tag{2.7}$$

Here, $s_{\alpha,r}$, $\alpha \in \{w, n\}$ are the phase residual saturations. The parameter $\theta \in [0.2, 3.0]$ is a result of the inhomogeneity of the medium. A highly heterogeneous porous medium is characterized by a large θ.

The capillary pressure $p_c = p_c(s_{w,e})$ is a function of the phase saturation. For the Brooks-Corey formulation,

$$p_c(s) = p_d s_{w,e}^{-1/\theta}. \tag{2.8}$$

Here, $p_d \geq 0$ is the constant entry pressure, needed to displace the fluid from the largest pore.

2.1.1 Wetting-phase-pressure/nonwetting-phase-saturation formulation

From the constitutive relations (2.3) and (2.4), we can rewrite the two-phase flow problem as a system of two equations with two unknowns p_w and s_n,

$$\begin{aligned} -\nabla \cdot (\lambda_t K \nabla p_w + \lambda_n K \nabla p_c - (\rho_w \lambda_w + \rho_n \lambda_n) K g) &= q_w + q_n, \\ \phi \frac{\partial s_n}{\partial t} - \nabla \cdot (\lambda_n K (\nabla p_w - \rho_n g)) - \nabla \cdot (\lambda_n K \nabla p_c) &= q_n. \end{aligned} \tag{2.9}$$

Here, $\lambda_t = \lambda_w + \lambda_n$ denotes the total mobility. The first equation of (2.9) is of elliptic type with respect to the pressure p_w. The type of the second equation of (2.9) is either nonlinear hyperbolic if $\frac{\partial p_c(s_n)}{\partial s_n} \equiv 0$ or degenerate parabolic if the capillary pressure is not neglected. The diffusion term might degenerate if $\lambda_n(s_n = 0) = 0$.

2.1.2 Boundary properties

In order to have a complete system we add appropriate boundary and initial conditions. Thus, we assume that the boundary of the system is divided into disjoint open sets $\partial\Omega = \bar{\Gamma}_D \cup \bar{\Gamma}_N$. We denote by n the outward normal to $\partial\Omega$.

$$s_n(x, 0) = s_n^0(x), p_w(x, 0) = p_w^0(x) \qquad \forall x \in \Omega, \tag{2.10}$$

$$p_w(x, t) = p_{w_D}(x, t), s_n(x, t) = s_{n_D}(x, t) \qquad \forall x \in \Gamma_D, \tag{2.11}$$

$$v_\alpha \cdot n = J_\alpha(x, t), J_t = \sum_{\alpha \in \{w,n\}} J_\alpha \qquad \forall x \in \Gamma_N. \tag{2.12}$$

Here, J_α, $\alpha \in \{w, n\}$ is the inflow. In order to make p_w uniquely determined the Dirichlet boundary Γ_D should be of positive measure.

3 Discretization

Let $\mathcal{T}_h = \{E\}$ be a family of non-degenerate, quasi-uniform, possibly non-conforming partitions of Ω consisting of N_h elements (quadrilaterals or triangles in 2d, tetrahedrons or hexahedrons in 3d) of maximum diameter h. Let Γ^h be the union of the open sets that coincide with internal interfaces of elements of $\mathcal{T}_h$. Dirichlet and Neumann boundary interfaces are collected in the set Γ^h_D and Γ^h_N. Let e denote an interface in Γ^h shared by two elements E_- and E_+ of $\mathcal{T}_h$; we associate with e a unit normal vector n_e directed from E_- to E_+. We also denote by $|e|$ the measure of e. The discontinuous finite element space is $\mathcal{D}_r(\mathcal{T}_h) = \{v \in \mathbb{L}^2(\Omega) : v_{|E} \in \mathcal{P}_r(E) \ \ \forall E \in \mathcal{T}_h\}$, where $\mathcal{P}_r(E)$ denotes $\mathbb{Q}_r$ (resp. $\mathbb{P}_r$) the space of polynomial functions of degree at most $r \geq 1$ on E (resp. the space of polynomial functions of total degree $r \geq 1$ on E). We approximate the pressure and the saturation by discontinuous polynomials of total degrees r_p and r_s respectively.
For any function $q \in \mathcal{D}_r(\mathcal{T}_h)$, we define the jump operator $[\![\cdot]\!]$ and the average operator $\{\cdot\}$ over the interface e:

$\forall e \in \Gamma^h, \quad [\![q]\!] := q_{E_-} - q_{E_+}, \quad \{q\} := \frac{1}{2}q_{E_-} + \frac{1}{2}q_{E_+},$

$\forall e \in \partial\Omega, \quad [\![q]\!] := q_{E_-}, \quad \{q\} := q_{E_-}.$

In order to treat the strong heterogeneity of the permeability tensor, we follow [16] and introduce a weighted average operator $\{\cdot\}_\omega$:

$\forall e \in \Gamma^h, \ \ \{q\}_\omega = \omega_{E_-} q_{E_-} + \omega_{E_+} q_{E_+},$

$\forall e \in \partial\Omega, \ \ \{q\}_\omega = q_{E_-}.$

The weights are $\omega_{E_-} = \frac{\delta_K^{E_+}}{\delta_K^{E_+}+\delta_K^{E_-}}$, $\omega_{E_+} = \frac{\delta_K^{E_-}}{\delta_K^{E_+}+\delta_K^{E_-}}$ with $\delta_K^{E_-} = n_e^T K_{E_-} n_e$ and $\delta_K^{E_+} = n_e^T K_{E_+} n_e$. Here, K_{E_-} and K_{E_+} are the permeability tensors for the elements E_- and E_+.

3.1 Semi discretization in space

The derivation of the semi-discrete DG formulation is standard (see [8], [16], [18]). First, we multiply each equation of (2.9) by a test function and integrate over each element, then we apply Green formula to obtain the semi-discrete weak DG formulation. Hence, the aforementioned formulation consists in finding the continuous in time approximations $p_{w,h}(\cdot,t) \in \mathcal{D}_{r_p}(\mathcal{T}_h)$, $s_{n,h}(\cdot,t) \in \mathcal{D}_{r_s}(\mathcal{T}_h)$ such that:

$$\mathcal{B}_h(p_{w,h}, \varphi; s_{n,h}) = l_h(\varphi) \qquad \forall \varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h), \forall t \in \mathcal{J}, \tag{3.1}$$

$$(\Phi \partial_t s_{n,h}, \psi) + c_h(p_{w,h}, \psi; s_{n,h}) + d_h(s_{n,h}, \psi) = r_h(\psi) \qquad \forall \psi \in \mathcal{D}_{r_s}(\mathcal{T}_h), \forall t \in \mathcal{J}. \tag{3.2}$$

The bilinear form $\mathcal{B}_h$ in the total fluid conservation equation (3.1) is expressed as:

$$\mathcal{B}_h(p_{w,h}, \varphi; s_{n,h}) = \mathcal{B}_{bulk,h} + \mathcal{B}_{cons,h} + \mathcal{B}_{sym,h} + \mathcal{B}_{stab,h}. \tag{3.3}$$

The first term $\mathcal{B}_{bulk,h}$ of (3.3) is the volume contribution:

$$\mathcal{B}_{bulk,h} := \mathcal{B}_{bulk,h}(p_{w,h}, \varphi; s_{n,h}) = \sum_{E \in \mathcal{T}_h} \int_E (\lambda_t K \nabla p_{w,h} + \lambda_n K \nabla p_{c,h} - (\rho_n \lambda_n + \rho_w \lambda_w) K g) \cdot \nabla \varphi. \tag{3.4}$$

The second term $\mathcal{B}_{cons,h}$, is the consistency term:

$$\begin{aligned} \mathcal{B}_{cons,h} := \mathcal{B}_{cons,h}(p_{w,h}, \varphi; s_{n,h}) = &- \sum_{e \in \Gamma^h \cup \Gamma^h_D} \int_e \{\lambda_t K \nabla p_{w,h}\}_\omega [\![\varphi]\!] \\ &- \sum_{e \in \Gamma^h \cup \Gamma^h_D} \int_e \{\lambda_n K \nabla p_{c,h}\}_\omega [\![\varphi]\!] \\ &+ \sum_{e \in \Gamma^h \cup \Gamma^h_D} \int_e \{(\rho_n \lambda_n + \rho_w \lambda_w) K g)\}_\omega [\![\varphi]\!]. \end{aligned} \tag{3.5}$$

 © by the authors, 2017

The third term $\mathcal{B}_{sym,h}$, is the symmetry term. Depending on the choice of ϵ we get different DG methods ($\epsilon = -1$ SIPG, $\epsilon = 1$ NIPG, $\epsilon = 0$ IIPG):

$$\begin{aligned}\mathcal{B}_{sym,h} := \mathcal{B}_{sym,h}(p_{w,h}, \varphi; s_{n,h}) =& \epsilon \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{\lambda_t K \nabla\varphi\}_\omega \cdot n_e [\![p_{w,h}]\!] \\ &+ \epsilon \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{\lambda_n K \nabla\varphi\}_\omega \cdot n_e [\![s_{n,h}]\!].\end{aligned} \tag{3.6}$$

The last term $\mathcal{B}_{stab,h}$ is the stability term:

$$\mathcal{B}_{stab,h} := \mathcal{B}_{stab,h}(p_{w,h}, \varphi) = \sum_{e\in\Gamma^h\cup\Gamma_D^h} \gamma_e^p \int_e [\![p_{w,h}]\!][\![\varphi]\!]. \tag{3.7}$$

The right hand side of the total fluid conservation equation (3.1) is a linear form including the Neumann and Dirichlet boundary conditions and the source terms.

$$\begin{aligned} l_h(\varphi) =& \int_\Omega (q_w + q_n)\varphi - \sum_{e\in\Gamma_N} \int_e J_t \varphi + \epsilon \sum_{e\in\Gamma_D^h} \int_e \lambda_t K \nabla\varphi \cdot n_e p_D \\ &+ \epsilon \sum_{e\in\Gamma_D^h} \int_e \lambda_n K \nabla\varphi \cdot n_e s_D + l_{stab}, \qquad \forall\varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h).\end{aligned} \tag{3.8}$$

Here, $l_{stab}(\varphi)$ is the stability term for the linear form:

$$l_{stab}(\varphi) = \sum_{e\in\Gamma_D^h} \gamma_e^p \int_e p_D \varphi. \tag{3.9}$$

Equation (3.2) is the discrete weak formulation of the nonwetting-phase conservation equation where the convection term $-\nabla\cdot(\lambda_n K(\nabla p_w - \rho_n g))$ might be approximated by an upwind discretization technique.

$$\begin{aligned} c_h(p_{w,h}, \psi; s_{n,h}) =& \sum_{E\in\mathcal{T}_h} \int_E (K\lambda_n(\nabla p_{w,h} - \rho_n g)) \cdot \nabla\psi - \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{K\lambda_n^\# \nabla p_{w,h}\}_\omega \cdot n_e [\![\psi]\!] \\ &+ \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{\rho_n K \lambda_n^\# g\}_\omega \cdot n_e [\![\psi]\!] + \epsilon \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{K\lambda_n^\# \nabla\psi\}_\omega \cdot n_e [\![p_{w,h}]\!],\end{aligned} \tag{3.10}$$

where $\lambda_n^\# = (1-\varrho)\lambda_{n,E} + \varrho\lambda_n^\uparrow$ and $\lambda_n^\uparrow$ is the upwind mobility:

$$\forall e \in \partial E_- \cap \partial E_+, \quad \lambda_n^\uparrow = \begin{cases} \lambda_{n,E_-} & \text{if } -K(\nabla p_w + \nabla p_c - \rho_n g)\cdot n \geq 0, \\ \lambda_{n,E_+} & \text{else.} \end{cases}$$

Hence depending on the value of $\varrho \in \{0,1\}$, we might use central differencing or upwinding of the mobility for internal interfaces.
The diffusion term $-\nabla\cdot(\lambda_n K \nabla p_c)$ is discretized by a bilinear form similar to that of (3.3).

$$\begin{aligned} d_h(s_{n,h}, \psi) =& \sum_{E\in\mathcal{T}_h} \int_E \lambda_n K \nabla p_{c,h} \cdot \nabla\psi - \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{\lambda_n K \nabla p_{c,h}\}_\omega \cdot n_e [\![\psi]\!] \\ &+ \epsilon \sum_{e\in\Gamma^h\cup\Gamma_D^h} \int_e \{\lambda_n K \nabla\psi\}_\omega \cdot n_e [\![s_{n,h}]\!] + \sum_{e\in\Gamma^h\cup\Gamma_D^h} \gamma_e^s \int_e [\![s_{n,h}]\!][\![\psi]\!].\end{aligned} \tag{3.11}$$

© by the authors, 2017

The right hand side r_h includes the Neumann and Dirichlet boundary condition and the nonwetting source term.

$$\begin{aligned} r_h(\psi) = & \int_\Omega q_n\psi - \sum_{e\in\Gamma_N} \int_e J_n\psi + \epsilon \sum_{e\in\Gamma_D^h} \int_e \lambda_n K\nabla\psi \cdot n_e p_D \\ & + \epsilon \sum_{e\in\Gamma_D^h} \int_e \lambda_n K\nabla\psi \cdot n_e p_c(s_D) + \sum_{e\in\Gamma_D^h} \gamma_e^s \int_e s_D\psi, \quad \forall\psi \in \mathcal{D}_{r_s}(\mathcal{T}_h). \end{aligned} \tag{3.12}$$

Remark 3.1 *The penalty terms γ_e^p and γ_e^s are discrete positive functions that take constant values on the interfaces. In order to ensure stability and convergence of the DG method, γ_e^p and γ_e^s must be chosen properly. This choice is especially crucial for strongly heterogeneous problems where parameters such as permeability, porosity, entry pressures can vary strongly, hence triggering a strong effect on the solution behavior. Following [8], we use in this work, unless specified otherwise, the penalty formulation as below.*

$$\gamma_e^p = C_p \frac{r_p(r_p + d - 1) \mid e \mid}{min(\mid E_- \mid, \mid E_+ \mid)}, \; C_p \geq 0 \tag{3.13}$$

and

$$\gamma_e^s = C_s \frac{r_s(r_s + d - 1) \mid e \mid}{min(\mid E_- \mid, \mid E_+ \mid)}, \; C_s \geq 0. \tag{3.14}$$

In the context of higher order discretization of large scale complex multiphase flow, the choice of proper basis functions is decisive for the computational efficiency and accuracy of the solver. In the sequel, we present the families of modal and nodal basis functions.

3.1.1 Modal basis The modal basis functions are sets of orthogonal polynomials w.r.t an appropriate inner product. They are also designed to have desirable properties such as hierachism, that is to say the basis for a given polynomial degree r includes the bases for polynomials degrees less than r. The use of hierarchical basis is essential for the prospect of higher order methods and local polynomial order adaptivity. The approximate solution $s_h^E(x,t)$ on each element E can be expressed as:

$$s_h^E(x,t) = \sum_{j=1}^{N_{loc}} \hat{s}_j^E(t)\psi_j(x), \;\; \forall E \in \mathcal{T}_h, \tag{3.15}$$

where the term $\{\hat{s}_j(t)\}_{j=1,\dots,N_{loc}}$ denotes the time dependent modal dofs and $\psi_j(x)$ is a d-dimensional polynomial basis. In the case of piecewise polynomials of total degree at most r, the local dimension N_{loc} is:

$$N_{loc} = \#\mathbb{P}_r = \frac{(r+d)!}{r!d!}. \tag{3.16}$$

In the case of piecewise polynomial of degree at most r in each variable the local dimension is:

$$N_{loc} = \#\mathbb{Q}_r = (r+1)^d. \tag{3.17}$$

A classical choice to generate modal basis functions $\psi_j(x)$ is to choose:

$$\psi_j(x) = P_{j+1}(x) - P_j(x), \quad j = 1, \dots, N_{loc}. \tag{3.18}$$

where P_j is the Legendre polynomial of degree j. It is also possible to use a Gramm-Schmidt procedure with the usual inner product to build an orthonormal basis from an initial monomial basis (see, e.g. [20]).

 © by the authors, 2017

3.1.2 Nodal basis The nodal approach is based on Lagrange polynomials with roots at a set of nodal points. Therefore, the local approximation is:

$$s_h^E(x,t) = \sum_{i=1}^{N_{loc}} \tilde{s}^E(x_i,t) l_i^E(x), \tag{3.19}$$

where $l_i^E(x)$ is the d-dimensional Lagrange polynomial based on the nodal set $\{x_i\}_{i=1,\dots,N_{loc}}$.
Following Heasthaven [17], it is possible to switch from modal to nodal and vice-versa.

$$\tilde{s} = V\hat{s}, \qquad V^\top l(x) = \psi(x), \qquad V_{i,j} = \psi_j(x_i), \tag{3.20}$$

where V is the Vandermonde matrix containing the evaluation of modal polynomials at the interpolation points. This transformation allows to evaluate efficiently higher dimensional Lagrange polynomials $l_i(x)$.

3.2 Fully coupled/Fully implicit DG scheme

The time interval $[0,T]$ is divided into N intervals $\Delta t_i = t_{i+1} - t_i$ as $0 = t_0 \le t_1 \le \cdots \le t_{N-1} \le t_N = T$. Let p_w^i and s_n^i be the numerical solutions at time t^i. We also denote $\lambda_\alpha^i = \lambda_\alpha(s_n^i)$, $p_c^i = p_c(s_n^i)$. The approximation $s_{n,h}^0$ is chosen as the $\mathbb{L}^2$ projection of the saturation $s_n(0)$. For the sake of simplicity and easier reading, we apply a first order Adams-Moulton (Backward Euler) time discretization and Interior Penalty DG for space discretization to the semi-discrete system (3.1) - (3.2):

$$\mathcal{B}_h(p_{w,h}^{i+1}, \varphi; s_{n,h}^{i+1}) = l_h(\varphi), \qquad \forall \varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h), \tag{3.21}$$

$$(\Phi \frac{s_{n,h}^{i+1} - s_{n,h}^i}{\Delta t}, \psi) + c_h(p_{w,h}^{i+1}, \psi; s_{n,h}^{i+1}) + d_h(s_{n,h}^{i+1}, \psi) = r_h(\psi), \qquad \forall \psi \in \mathcal{D}_{r_s}(\mathcal{T}_h), \tag{3.22}$$

$$(s_{n,h}^0, \zeta) = (s_n^0, \zeta), \qquad \forall \zeta \in \mathcal{D}_{r_s}(\mathcal{T}_h). \tag{3.23}$$

(3.21)-(3.23) leads to a large, nonlinear system of algebraic equations written in the form:

$$G(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) = \begin{pmatrix} G^{r_p}(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) \\ G^{r_s}(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) \end{pmatrix} = 0, \tag{3.24}$$

where $\bar{p}_{w,h}^{i+1} = (p_{\underline{E}})_E$ and $\bar{s}_{w,h}^{i+1} = (s_{\underline{E}})_E$ are vectors of unknowns for $p_{w,h}^{i+1}$ and $s_{n,h}^{i+1}$.
The system is solved by using a Newton-Raphson method.

$$J_G(\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r})\delta^{r+1} = -G(\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r}), \tag{3.25}$$

$$(\bar{p}_w^{i+1,r+1}, \bar{s}_n^{i+1,r+1}) = \delta^{r+1} + (\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r}). \tag{3.26}$$

Here, r denotes the rth Newton iterate and for a coupled system such as (3.25), the Jacobian J_G is:

$$J_G = \begin{pmatrix} J_{pp} & J_{ps} \\ J_{sp} & J_{ss} \end{pmatrix} = \begin{pmatrix} \frac{\partial G^{r_p}}{\partial p} & \frac{\partial G^{r_p}}{\partial s} \\ \frac{\partial G^{r_s}}{\partial p} & \frac{\partial G^{r_s}}{\partial s} \end{pmatrix}.$$

4 Adaptivity strategy

For the considered DG discretization of porous media two-phase flow problem, derivation of error estimates based on rigorous a-posteriori error estimates is out of the scope of this paper. Hence, we use heuristic indicators which depend on the local gradient of the nonwetting-phase-saturation s_n measured in the $\mathbb{L}^2$ norm. We define on each element E of the mesh, the indicator η_E^i at time step i, such that:

$$\eta_E^i = \|\nabla s_n^i\|_{L^2(E)}, \qquad \forall E \in \mathcal{T}_h. \tag{4.1}$$

Each element whose indicator η_E^i is greater than a treshold value $\eta_{Tol} \ge 0$ is refined.

© by the authors, 2017

5 Implementation using Dune-Fem

This section is dedicated to an overview of the implementation of the DG two-phase flow simulator based on Dune-Fem. For a more in-depth description of the Dune-Fem interface, we refer to [12].

5.1 Library requirements

Dune-twophaseDG needs the Dune core modules Dune-Common, Dune-Grid, Dune-Localfunctions, Dune-Istl at version 2.3 (or later) and the Dune-Fem module at version 1.4 (or later). For performing *h*-adaptivity, one has to use adaptive grids such as Alugrid_Cube or Alugrid_Simplex from the library Dune-Alugrid, non-adaptive grids such as YaspGrid or SGrid do not provide local adaptivity.

5.2 Structure and code description

We describe the structure of the directory of Dune-twophaseDG in terms of subdirectories, header files and executable files. The following subdirectories are within the module:

- *CMake*: configuration options for building the module while using Cmake.
- *doc*: doxygen documentation.
- *dune*: header files.
- *src*: source files for the numerical examples.

5.2.1 The directory *dune* The directory *dune* contains different subdirectories:

- *algorithm*: contains the header files algorithm.hh, femscheme.hh, phaseflowscheme.hh, probleminterface.hh and temporalprobleminterface.hh.
- *estimator*: contains the header file estimator.hh providing a heuristic estimator for the numerical solution.
- *models*: contains the header file phaseflowmodel.hh.
- *operator*: contains the header files operator.hh providing the classes which build the discrete stiffness matrix and the right hand side. The header file newtoninvop.hh provides the Newton inverse operator.

5.2.2 The directory *examples* The directory *examples* contains the *lenspb* folder holding the infiltration problem header files lenspbbndmodel.hh, lenspbinitialdata.hh, lenspbmodel.hh and lenspbphysicalparmodel.hh.

5.2.3 The directory *src* The *src* directory contains the source files for the different numerical modules:

- *3dlens*: 3d infiltration problem with gravity forces and capillarity effects,
- *2dlens*: 2d infiltration problem with gravity forces and capillarity effects.

In each test, we have a source file for the main program (i.e. 3dlens.cc, 2dlens.cc).

 © by the authors, 2017

5.3 Features of the implementation

The implementation of the discrete formulation (3.21)-(3.23) is realized in a similar fashion as most of the tutorial examples from the Dune-Fem-howto. Starting from the included heat DG problem, we extend it to a system of two equations and two unknowns and we add the non-linear formulation. First, we describe the PDE by a class phaseflowmodel, which serve as an interface for general test cases. This is also the interface used to implement the operator. In the phaseflowmodel (see Code 1), the methods wetting_Pressure_DiffusiveFlux() for the wetting pressure flux, capillary_Pressure_DiffusiveFlux() for the capillary pressure flux and gravity_Flux() for the gravity flux will return the terms multiplied with ∇v. The source() method will return the parts of the operator multiplied with v. In order to handle the non-linearity, we also add the methods linSource() (resp. linWetting_Pressure_DiffusiveFlux(), linCapillary_Pressure_DiffusiveFlux() and linGravity_Flux()) returning the linearization of the source term (resp. flux terms). The DNAPL infiltration test case has its own model lenspbmodel which derives from the phaseflowmodel. The initial and boundary data are specified respectively in lenspbinitialdata.hh and lenspbbndmodel.hh. The lenspbphysicalparmodel class specifies the different physical properties of the lens problem such as the mobility and the capillary pressure function formulations.

```
template< class FunctionSpace, class GridPart, class PhysParModel >
struct PhaseFlowModel
{
    template< class Entity, class Point >
  void source ( const Entity &entity,
                const Point &x,
                const RangeType &value,
                RangeType &flux ) const;

  template< class Entity, class Point >
  void linSource ( const RangeType& valueUn,
                   const Entity &entity,
                   const Point &x,
                   const RangeType &value,
                   RangeType &flux ) const;


  //! return the wetting pressure flux
  template< class Entity, class Point >
  void wetting_Pressure_DiffusiveFlux ( const Entity &entity,
                                        const Point &x,
                                        const RangeType &value,
                                        const JacobianRangeType &gradient,
                                        JacobianRangeType &flux,
                                        const double lambda_n_upw=1,
                                        const bool &useupw=false,
                                        const bool &buildRhs=false ) const;

  //! return the linearized wetting pressure flux
  template< class Entity, class Point >
  void linWetting_Pressure_DiffusiveFlux ( const RangeType &valueUn,
                                           const JacobianRangeType &gradientUn,
                                           const Entity &entity,
                                           const Point &x,
                                           const RangeType &value,
                                           const JacobianRangeType &gradient,
                                           JacobianRangeType &flux,
                                           const double lambda_n_upw = 1,
                                           const double gradnonwetmob_upw = 1,
                                           const bool useupw=false ) const;
}
```

Code 1: Excerpt from phaseflowmodel.hh.

The assembly process of the operator and the right hand side is done in the file operator.hh. The class FlowOperator is derived from the Dune::Operator class. This is why we need to override the operator() method. In order to build the jacobian, we introduce a class DifferentiableFlowOperator (see Code 2) which derives from the FlowOperator and from the interface class DifferentiableOperator:

© by the authors, 2017

```
template< class JacobianOperator, class Model >
struct DifferentiableFlowOperator
  : public FlowOperator< typename JacobianOperator::DomainFunctionType, Model >,
    public Dune::Fem::DifferentiableOperator< JacobianOperator >
```

Code 2: DifferentiableFlowOperator.

In order to build the operator, we iterate over the intersections of the elements. For each intersection, we evaluate the local functions on the elements on both sides of the intersection by using a `FaceQuadratureType::INSIDE` for the element and `FaceQuadratureType::OUTSIDE` for the neighboring element. Code 3 shows the assembly process of the operator where the method `volumetricPart()` (line 24) computes the local contribution from each element and `internal_Bnd_terms()` (line 34) computes local contribution from interfaces and boundaries.

```
template< class DiscreteFunction, class Model >
void FlowOperator< DiscreteFunction, Model >
::operator() ( const DiscreteFunctionType &u, DiscreteFunctionType &w ) const
{
  // clear destination
  w.clear();
  // get discrete function space
  const DiscreteFunctionSpaceType &dfSpace = w.space();

  // iterate over grid
  const IteratorType end = dfSpace.end();
  for( IteratorType it = dfSpace.begin(); it != end; ++it )
  {
    // get entity (here element)
    const EntityType &entity = *it;
    // get elements geometry
    const GeometryType &geometry = entity.geometry();
    // get local representation of the discrete functions
    const LocalFunctionType uLocal = u.localFunction( entity );
    LocalFunctionType wLocal = w.localFunction( entity );
    // obtain quadrature order
    const int quadOrder = uLocal.order() + wLocal.order();
    //Computing local contribution from elements
    volumetricPart(entity, quadOrder, geometry, uLocal, wLocal);

    if ( ! dfSpace.continuous() )
    {
      const IntersectionIteratorType iitend = dfSpace.gridPart().iend( entity );
      // looping over intersections
      for( IntersectionIteratorType iit = dfSpace.gridPart().ibegin( entity ); iit != iitend; ++iit
      )
      {
        const IntersectionType &intersection = *iit;
        //Computing local contribution from interfaces and boundaries
        internal_Bnd_terms(entity, quadOrder, geometry, intersection, dfSpace, uLocal, u, wLocal);
      }
    }
  }
  // communicate data (in parallel runs)
  w.communicate();
}
```

Code 3: Operator building.

The selection of the type of discrete function space is done in the `femscheme` class. The discrete function space depends on the `GridPartType` and the `FunctionSpace` (see Code 4). It allows to choose the polynomial order by setting the parameter `POLORDER`, hence permitting the use of higher order polynomials without any further changes in the code. For the sake of simplicity and usability, the code only supports the case $r_p = r_s =$ `POLORDER`.

```
 //! choose type of discrete function space and the polynomial order POLORDER
#if USE_LAG_DG
  typedef Dune::Fem::LagrangeDiscontinuousGalerkinSpace< FunctionSpaceType, GridPartType, POLORDER
     > DiscreteFunctionSpaceType;
#else
  typedef Dune::Fem::DiscontinuousGalerkinSpace< FunctionSpaceType, GridPartType, POLORDER >
     DiscreteFunctionSpaceType;
#endif
```

Code 4: Type of discrete function space and polynomial order.

 © by the authors, 2017

In order to achieve an adaptive scheme, we implement an `estimator` class which supports a method `mark()` (see Code 5) to mark the elements for the next refinement step. Our marking strategy consist in looping over the mesh and selecting for refinement all elements where the $\mathbb{L}^2$ norm of the saturation gradient is larger than a certain tolerance η_{Tol}. The value of the η_{Tol} can be specified in the parameter file with the variable `phaseflow.tolerance`.

```
 //! mark all elements due to given tolerance
bool mark ( const double tolerance ) const
{
  int marked = 0;
  // loop over all elements
  const IteratorType end = dfSpace_.end();
  for( IteratorType it = dfSpace_.begin(); it != end; ++it )
    {
      const ElementType &entity = *it;


      const Dune::ReferenceElement< double, dimension > &refElement
      = Dune::ReferenceElements< double, dimension >::general( entity.type() );
      RangeType val;
      // evaluate the phase field at the barycentre (note
      // refElement.position(0,0) is the barycentre in local coordinates)
      double markVal;
      JacobianRangeType grad;
      uh_.localFunction(entity).jacobian(refElement.position(0,0),grad);
      markVal = grad[1].two_norm();


      if( markVal > tolerance )
      {
        // make sure grid is not overly refined...
        // maxLevel_ is the maximum level of refinement allowed
        if (entity.level() < maxLevel_)
        {
          // mark entity for refinement
          grid_.mark( 1, entity );
          // grid was marked
          marked = 1;
        }
      }
     else
     {
        // mark for coarsening
        grid_.mark( -1, entity );
     }
    }
  // get global max
  marked = grid_.comm().max( marked );
  return bool(marked);
}
```

Code 5: Excerpt from `estimator.hh`.

In the main function (see Code 6), we first initialize `MPI`, then read the parameters and construct the grid based on the grid implementation provided in `CMakeLists.txt`. After initializing the grid, we get an instance of the class `Algorithm` containing the algorithm (line 26). After that, the function compute is executed in line 30.

```
int main ( int argc, char **argv )
  try
    {
      // initialize MPI, if necessary
      Dune::Fem::MPIManager::initialize( argc, argv );
      // append overloaded parameters from the command line
      Dune::Fem::Parameter::append( argc, argv );
      // append possible given parameter files
      for( int i = 1; i < argc; ++i )
       Dune::Fem::Parameter::append( argv[ i ] );
      // append default parameter file
      Dune::Fem::Parameter::append( "../data/parameter" );
      // type of hierarchical grid
      typedef Dune::GridSelector::GridType  HGridType ;
      typedef Algorithm< HGridType > AlgorithmType;
      // create grid from DGF file
      const std::string gridkey = Dune::Fem::IOInterface::defaultGridKey( HGridType::dimension );
      const std::string gridfile = Dune::Fem::Parameter::getValue< std::string >( gridkey );
```

© by the authors, 2017

```
19      // the method rank and size from MPIManager are static
20      if( Dune::Fem::MPIManager::rank() == 0 )
21       std::cout << "Loading macro grid: " << gridfile << std::endl;
22      // construct macro using the DGF Parser
23      Dune::GridPtr< HGridType > gridPtr( gridfile );
24      HGridType& grid = *gridPtr ;
25
26      AlgorithmType myalgorithm (grid);
27      // Compute algorithm
28      Dune::Timer computetimer;
29      //Compute the algorithm
30      myalgorithm.compute();
31      const double compuTime = computetimer.elapsed();
32
33      ...
34
35      return 0;
36    }
```

Code 6: main function of 2dlens.cc.

In the `compute()` method of the class `Algorithm` (Code 7), we initialize two model instances which are passed on to the `Scheme` class. Here, two elliptic operators are constructed and used to evolve the solution from one time level to the next. The `TimeProvider` class is used to handle time dependency.

```
1      // create time provider
2     Dune::Fem::GridTimeProvider< HGridType > timeProvider( grid_ );
3     // we want to solve the problem on the leaf elements of the grid
4     GridPartType gridPart(grid_);
5     // type of the mathematical model used
6     ProblemType problem( timeProvider ) ;
7     // implicit model for left hand side
8     ModelType implicitModel( problem, gridPart, true );
9     // explicit model for right hand side
10    ModelType explicitModel( problem, gridPart, false );
11    // create scheme
12    SchemeType scheme( gridPart, implicitModel, explicitModel );
13    //! input/output tuple and setup datawritter
14    IOTupleType ioTuple( &(scheme.solution())) ; // tuple with pointers
15    DataOutputType dataOutput( grid_, ioTuple );
16
17    const bool local_adapt = Dune::Fem::Parameter::getValue< bool >("phaseflow.local_adapt", false
       );
18    const double endTime  = Dune::Fem::Parameter::getValue< double >( "phaseflow.endtime", 3.0 );
19    const double dtreducefactor = Dune::Fem::Parameter::getValue< double >("phaseflow.
       reducetimestepfactor", 1 );
20    double timeStep = Dune::Fem::Parameter::getValue< double >( "phaseflow.timestep", 0.00125 );
21    double tolerance = Dune::Fem::Parameter::getValue< double >("phaseflow.tolerance", 0.5 );
22
23    int step=1;
24    timeStep *= pow(dtreducefactor,step);
25    // initialize with fixed time step
26    timeProvider.init( timeStep ) ;
27    scheme.initialize();
28    // write initial solve
29    dataOutput.write( timeProvider );
30
31
32    // time loop, increment with fixed time step
33    for( ; timeProvider.time() < endTime; timeProvider.next( timeStep ) )
34      {
35        std::cout << "t=" << timeProvider.time() << std::endl;
36
37        if (local_adapt)
38         {
39           // mark element for adaptation
40           scheme.mark( tolerance );
41           // adapt grid
42           scheme.adapt();
43           scheme.prepare();
44           scheme.solve(true);
45           scheme.postpro();
46         }
47  ...
48      }
```

Code 7: Excerpt from `algorithm.hh`.

 © by the authors, 2017

5.4 Input & Output files

5.4.1 Input files

We use parameter files (see Code 8) to set parameters for the simulation.

```
# GENERAL ########################
#--------


#### Parameters for output ######
phaseflow.computeEOC: 0
# prefix data files
fem.io.datafileprefix: 2dtwophase
# save every i-th step
fem.io.savestep: 40.0e00


# specify directory for data output (is created if not exists)
fem.prefix: ../Output2D/Deg3/LagBasis


# upwinding of advective term
phaseflow.with_upw: false


#local adaptivity
phaseflow.local_adapt: true


# tolerance for estimator
phaseflow.tolerance: 5


#number of level of refinement
phasefield.maxlevel:2


#DG penalty
phaseflow.penaltypress: 1e-2
phaseflow.penaltysat: 1e-3


#DG method NIPG=-1 SIPG=1 IIPG=0
phaseflow.DGeps: 1


##################################
```

Code 8: Excerpt from the parameter file.

The input files are read in by the compiled program. Thus values can be modified at runtime (see Code 9).

```
// append default parameter file
Dune::Fem::Parameter::append( "../data/parameter" );
```

Code 9: Loading of the input file.

The `Dune::Parameter` singleton parses the given parameter file line by line. Code 10 shows how to use `Dune::Parameter`. The method `getValue()` expects two parameters. The first one is a `std::string`. The last one is a default value that will be used if the value of the parameter is not provided in the input file. This last parameter is optional.

```
const bool local_adapt = Dune::Fem::Parameter::getValue< bool >("phaseflow.local_adapt", false );
const double endTime  = Dune::Fem::Parameter::getValue< double >( "phaseflow.endtime", 3.0 );
double timeStep = Dune::Fem::Parameter::getValue< double >( "phaseflow.timestep", 0.00125 );
```

Code 10: Example of the `getValue()` method utilisation.

The grid type can either be specified directly or obtained from the `GridSelector`. The type is then specified during the make or `CMake` procedure.

© by the authors, 2017

```
add_definitions(
-DALUGRID_CUBE
-DPOLORDER=3
-DGRIDDIM=2
-DWORLDDIM=2
-DWANT_ISTL=0
-DUSE_LAG_DG=1
)
add_executable(2dlens 2dlens.cc)
add_dune_alugrid_flags(2dlens)
```

For more in-depth information on the input files we refer to the DUNE documentation [12].

5.4.2 Output files

The output is handled by the `DataOutput` class (see Line 4, Code 11) and a tuple holding pointers to `DiscreteFunction` objects is passed as a parameter. The generated vtu files are exported into the directory specified in the parameter file by `fem.prefix` (see Line 14, Code 8).

```
// type of input/output
typedef Dune::tuple< DiscreteFunctionType* > IOTupleType;
// type of the data writer
typedef Dune::Fem::DataOutput< HGridType, IOTupleType > DataOutputType;
IOTupleType ioTuple( &(scheme.solution())) ; // tuple with pointers
DataOutputType dataOutput( grid, ioTuple, DataOutputParameters( step ) );
```

Code 11: Output handling.

6 Numerical simulations

In this section we present some numerical tests for the presented DG scheme. Unless specified otherwise, all test cases are implemented with either the SIPG or the IIPG method. In order to ensure second order accuracy, we employ a central differencing of the mobility for internal interfaces thus following a similar approach to that of Rivière et al. [15]. We do not use any kind of slope limiting or upwinding techniques. The linear solver used is GMRES and we do not use any preconditioner. The maximal polynomial order employed for the 2d problem is $r_p = r_s = 3$. Although it is possible to use higher polynomial order (quartics and quintics), the schemes become computationally expensive in terms of both storage and CPU time for practical use.

6.1 Test Case 1: A vertical DNAPL infiltration Flow over a low permeability lens

A container is filled with two kinds of sand and saturated with water with density $\rho_w = 1000\ Kg/m^3$ and viscosity $\mu_w = 1 \times 10^{-3}\ Kg/m\ s$. The DNAPL considered in the experiment is Tetrachloroethylene with density $\rho_n = 1460\ Kg/m^3$ and viscosity $\mu_n = 9 \times 10^{-4}\ Kg/m\ s$.

Brooks-Corey's constitutive relations are used for the capillary pressure and the relative permeabilities. Discretization of the system is performed by Interior Penalty DG methods with a fully implicit/fully coupled approach. All test cases in this section include gravitational forces and capillary pressure effects. We use `ALUCubeGrid` for the test cases, it implements the `Dune GridInterface` for 3d hexahedral meshes. The grids can be locally adapted (non-conforming) and used in parallel computations [1].

 © by the authors, 2017

6.1.1 $2d$ **infiltration problem** We consider here a two-dimensional DNAPL infiltration problem with different sand types. The bottom of the reservoir is impermeable for both phases. Hydrostatic conditions for the pressure p_w and homogeneous Dirichlet conditions for the saturation s_n are prescribed at the left and right boundaries. A flux of $J_n = -5.137 \times 10^{-5}\ m\ s^{-1}$ of the DNAPL is infiltrated into the domain from the top. Detailed boundary conditions are specified in Table 2 and Figure 1. Initial conditions where the domain is fully saturated with water and hydrostatic pressure distribution are considered (i.e. $p_w^0 = (0.65 - y) \cdot 9810$, $s_n^0 = 0$). The initial mesh consists of 600 quadrilateral elements. We choose a time step of size $\Delta t = 5\ s$. The final time is $T = 2000\ s$. We consider a Newton solver tolerance $newtTol = 3 \times 10^{-7}$ and a linear solver tolerance $linabstol = 2.7 \times 10^{-7}$.

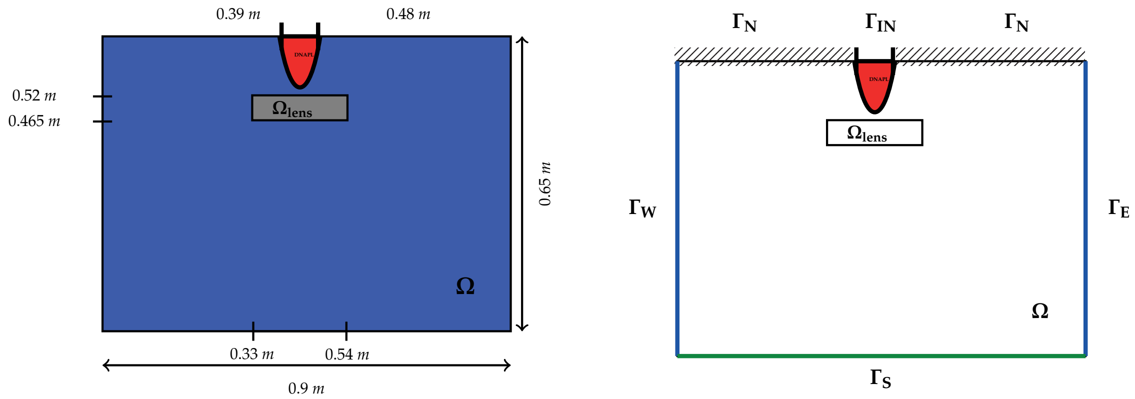


Figure 1: Geometry and boundary conditions for the DNAPL infiltration problem.

	Ω_{lens}	$\Omega \backslash \Omega_{lens}$
Φ [-]	0.39	0.40
k [m^2]	6.64×10^{-16}	6.64×10^{-11}
S_{wr} [-]	0.1	0.12
S_{nr} [-]	0.00	0.00
θ [-]	2.0	2.70
p_d [Pa]	5000	755

Table 1: Parameters.

Γ_{IN}	$J_n = -5.137 \times 10^{-5}$, $J_w = 0$
Γ_N	$J_n = 0.00$, $J_w = 0.00$
Γ_S	$J_w = 0$, $J_n = 0.00$
$\Gamma_E \cup \Gamma_W$	$p_w = (0.65 - y) \cdot 9810$, $s_n = 0$

Table 2: Boundary conditions.

Figure 2 and Figure 3 show the numerical results for the IIPG scheme with polynomial order $r_s = r_p = 3$ combined with first (resp. second) order Adams-Moulton method time discretization. We use here Lagrange DG space. The implementation of the Lagrange DG space is done by the mean of a Vandermonde matrix operating the transformation from spectral to physical space.

It took 520 s for the DNAPL to reach the lens and to spread out in the horizontal direction until reaching the edge of the lens. Afterwards the nonwetting front propagates down the sides of the lens. However as expected for advection dominated problems, we witness severe undershoots in the vicinity of the free boundary. The local h-adaptivity allows us to reduce those undershoots to small values. Figure 4 and Figure 5 show a comparison between the modal and Lagrange DG schemes. We witness smeared fronts for the orthonormal monomial basis unless we use small values of penalisation. The shape of the front for the Lagrange basis are less diffusive for large values of the penalisation parameter. Table 3 throws light upon the columns labels used in the numerical results. In Table 4, we provide details of the simulation including total computation times. As expected, the total computation time increases substantially with higher order polynomial degree. One can't help but notice that almost 80% (resp. 70%) of the total computing time is spent building the jacobian matrix for the DG/$\mathbb{Q}_3$ AM1 (resp. DG/$\mathbb{Q}_3$ AM2).

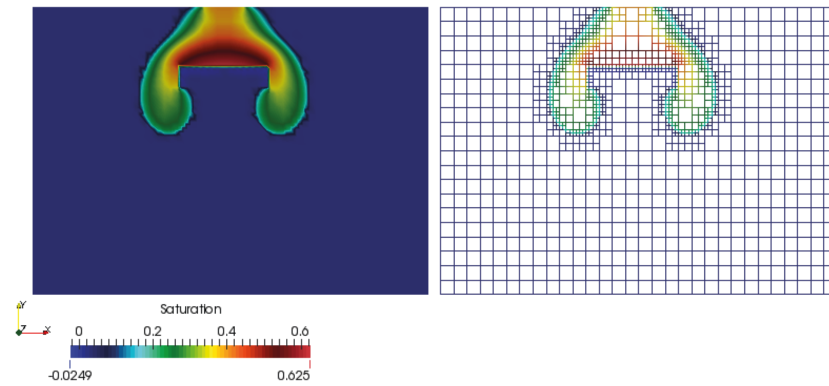


Figure 2: DNAPL saturation distribution after 2000 s (left), mesh distribution (right). Polynomial order $p = 3$.

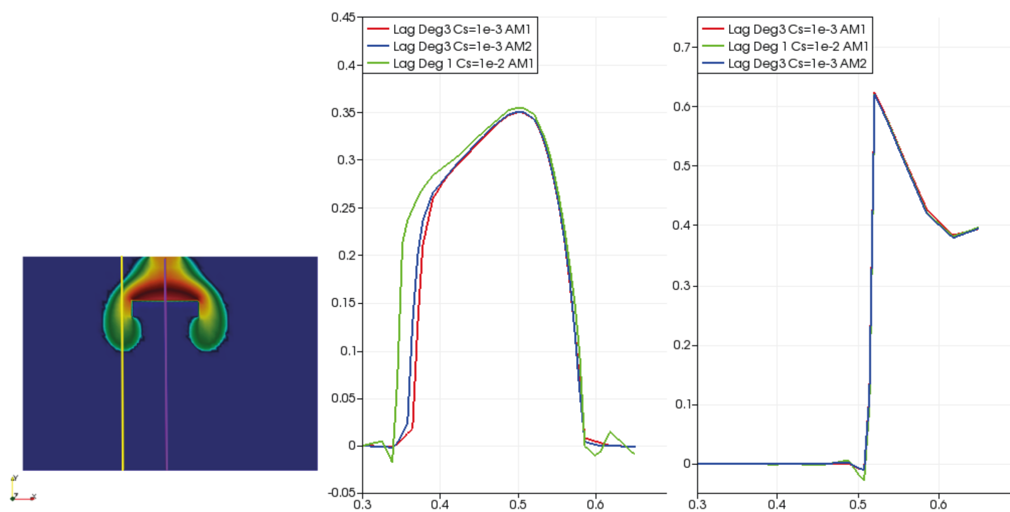


Figure 3: Comparison of non-wetting-phase saturation at T=2000 s. Center, profile along the line x=0.3 m (yellow vertical line of the left figure). Right, profile along the line x=0.45 m (violet vertical line of the left figure).

DG/$\mathbb{Q}_1$ AM1	Piecewise linear Lagrange DG combined with first order Adams-Moulton
DG/$\mathbb{Q}_3$ AM1	Piecewise cubic Lagrange DG combined with first order Adams-Moulton
DG/$\mathbb{Q}_3$ AM2	Piecewise cubic Lagrange DG combined with second order Adams-Moulton
Avg nb lin iter / Newton cycle	Average number of linear iterations per Newton cycle
Avg assem time / lin iter	Average time to assemble the Jacobian matrix per linear iteration
Avg inv time / lin iter	Average time to invert the Jacobian matrix per linear iteration

Table 3: Notation in result representation.

	DG/$\mathbb{Q}_1$ AM1	DG/$\mathbb{Q}_3$ AM1	DG/$\mathbb{Q}_3$ AM2
Avg nb lin iter / Newton cycle	486.869	310.574	517.365
Avg assem time / lin iter [sec]	0.75	9.78	9.25
Avg inv time / lin iter [sec]	0.21	2.67	4.03
Total cpu time [sec]	555.595	9669.63	10609.2

Table 4: Runtime overview for the 2d DNAPL infiltration problem.

 © by the authors, 2017

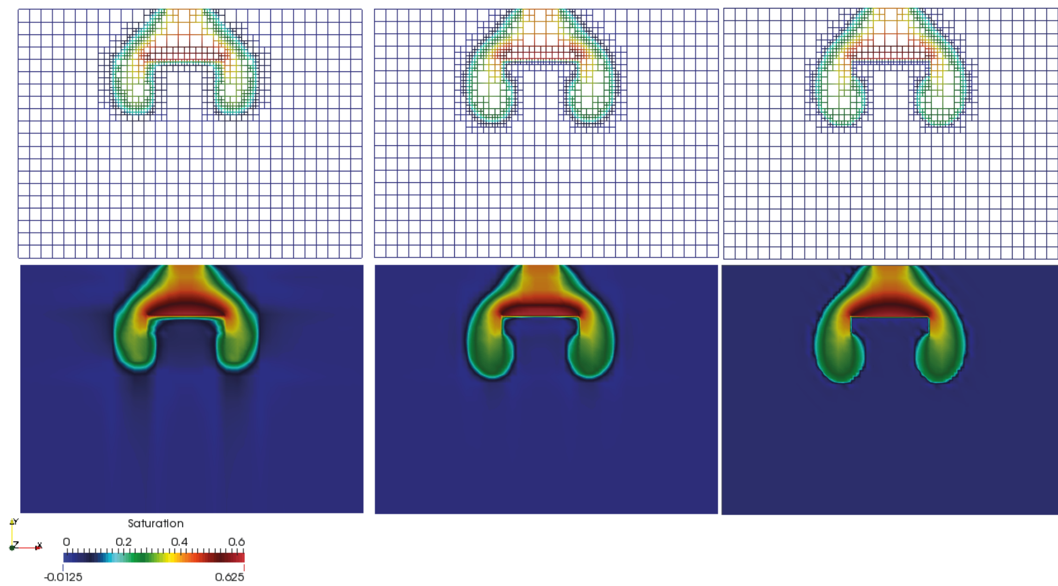


Figure 4: Contour plot of DNAPL saturation after 2000 s for modal orthonormal basis with $C_s = 1e-2$ (left column), modal orthonormal basis with $C_s = 1e-3$ (center column) and Lagrange basis with $C_s = 1e-2$ (right column). Polynomial order $p = 1$.

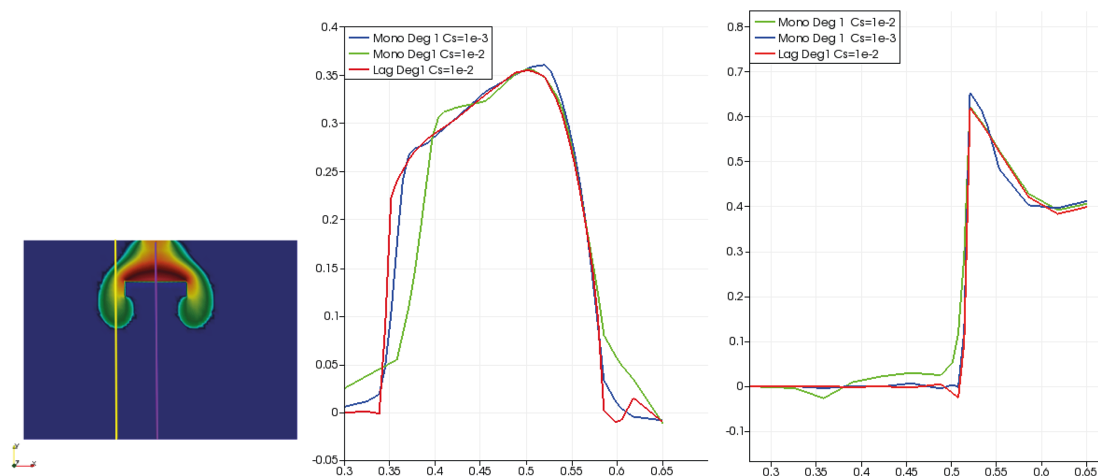


Figure 5: Comparison of non-wetting-phase saturation at T=2000 s. Center, profile along the line x=0.3 m (yellow vertical line of the left figure). Right, profile along the line x=0.45 m (violet vertical line of the left figure).

6.1.2 $3d$ **infiltration problem** In this section, we extend the previous results to the three-dimensional case. We also consider different sand types with different permeabilities and different entry pressures. The bottom of the reservoir is impermeable for both phases. Hydrostatic conditions for the pressure p_w and homogeneous Dirichlet conditions for the saturation s_n are prescribed at the lateral boundaries. A flux of $J_n = -1.712 \times 10^{-4}\ m\, s^{-1}$ of the DNAPL is infiltrated into the domain from the top. The initial `ALUCubeGrid` mesh consist of $10 \times 10 \times 10$ hexahedral elements and resolves the interfaces between regions with different permeabilities. 60 time steps of length $\Delta t = 60\ s$ are computed (final time $T = 3600\ s$). This grid is locally adapted (non-conforming).

© by the authors, 2017

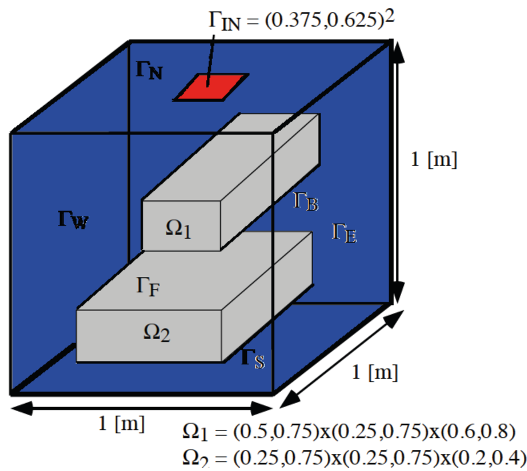


Figure 6: Geometry of the domain for the 3d DNAPL infiltration problem.

	Ω_1	Ω_2	$\Omega\backslash\Omega_1 \cap \Omega\backslash\Omega_2$
Φ [-]	0.39	0.39	0.40
k [m^2]	6.64×10^{-16}	6.64×10^{-15}	6.64×10^{-11}
S_{wr} [-]	0.1	0.1	0.12
S_{nr} [-]	0.00	0.00	0.00
θ [-]	2.0	2.0	2.70
p_d [Pa]	5000	5000	755

Figure 7: 3d problem parameters.

Figure 8 illustrates the evolution of the nonwetting saturation during the simulation. We show results at 3600 s of simulation time. As we increase the polynomial order, we notice undershoots in the vicinity of the front of the propagation and a sharp discontinuity in the solution at the lenses interfaces.

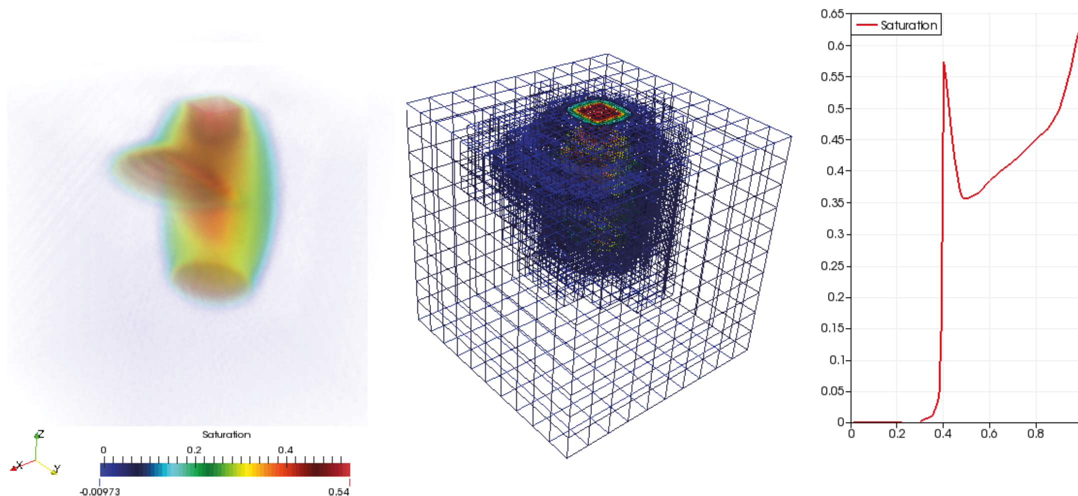


Figure 8: Contour plot of saturation distribution after 3600 s of DNAPL injection in a depth of 1 m (left column), mesh distribution (center column) and saturation profile along the line ((0.45,0.45,0);(0.45,0.45,1))(right column).

7 Conclusion & Outlook

In this work, we presented a discontinuous Galerkin scheme for incompressible, immiscible two-phase flow in strongly heterogeneous porous media with gravity forces and discontinuous capillary pressures. Higher-order polynomials up to piecewise cubics are implemented. The different test cases considered depict convection dominated problems such as DNAPL infiltration in an initially water saturated reservoir. The oscillations appearing in the the vicinity of the front of the propagation are reduced with the local mesh refinement. Undoubtedly this fully implicit DG requires further theoretical and numerical research. DG schemes such as Compact Discontinuous

 © by the authors, 2017

Galerkin 2 (CDG2)[9], which are less sensitive to the penalty parameter value seem to be suitable candidates for our models. Derivation of more rigorous a posteriori estimates that can robustly estimate both temporal and spatial error are of dire interest for more efficient adaptive algorithms. Effective local slope limiters are also needed in order to avoid the undershoots and overshoots witnessed in the vicinity of the front of the propagation.

Apendix A

Building the library

We present here the main steps to create a local working installation of `Dune-twophaseDG`.

- Create a DuneWorkspace directory.

```
1 $ mkdir DuneWorkspace && cd DuneWorkspace
```

- Checkout the latest (stable) core modules from the Dune project homepage.

```
1 $ for MOD in common geometry grid localfunctions istl; do
2 $ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-$MOD.git
3 $ done
```

- Checkout the latest (stable) version of `Dune-Fem`.

```
1 $ git clone -b releases/2.4 https://users.dune-project.org/repositories/projects/dune-fem.git
```

- Checkout `Dune-Alugrid` (required for the local grid adaptivity).

```
1 $ git clone -b releases/2.4  https://users.dune-project.org/repositories/projects/dune-alugrid.git
```

- Checkout `Dune-twophaseDG`.

```
1 $ git clone   https://gitlab.dune-project.org/birane.kane/dune-twophaseDG.git
```

Remark 7.1 *You can also download and unpack a* `Dune-twophaseDG` *tarball to a folder in your file system and extract the content of the tar files. Make sure that the extracted* `Dune-twophaseDG` *is in the DuneWorkspace directory.*

- Configure and compile the library by typing the following command in the DuneWorkspace directory.

```
1 $ cp dune-twophaseDG/scripts/opts/cmake.opts ./
2 $ ./dune-common/bin/dunecontrol --opts=cmake.opts all
```

Run of a test application

We assume in this section that the compilation of all required libraries has been completed in accordance with the description given in the previous section. The numerical model of `Dune-twophaseDG` are compiled in a build-folder (default: `build-cmake`) and tested in the test subfolder. For example, to run the 3d lens problem:

```
1 $ cd build-cmake/src/test/lenspb/3dlens
2 $ make 3dlens
3 $ ./3dlens -parameterFile ./../data/parameter3d
```

The parameter file specifies that all important parameters (like first time-step size, end of simulation and location of the grid file) can be found in a text file in the data directory with the name param*. The simulation starts and produces some .vtu output files and also a .pvd file in the folder `build-cmake/src/test/lenspb/Output3D`.

Remark 7.2 *All the test cases presented in this work can be executed with the following command.*

```
1 $ cd build-cmake/src/test/lenspb
2 $ source ../../../../scripts/allnumtest.sh
```

Acknowledgements

The author would like to acknowledge Dr. Bernd Flemisch and Dr. Claus J. Heine for valuable suggestions. Furthermore, the author would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart.

References

[1] M. Alkämper, A. Dedner, R. Klöfkorn, and M. Nolte. The dune-alugrid module. *arXiv preprint arXiv:1407.6954*, 2014.

[2] D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM journal on numerical analysis*, 19(4):742–760, 1982.

[3] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM journal on numerical analysis*, 39(5):1749–1779, 2002.

[4] G. A. Baker. Finite element methods for elliptic equations using nonconforming elements. *Mathematics of Computation*, 31(137):45–59, 1977.

[5] F. Bassi and S. Rebay. A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier–stokes equations. *Journal of computational physics*, 131(2):267–279, 1997.

[6] P. Bastian. Numerical computation of multiphase flow in porous media. 1999.

[7] P. Bastian. Higher order discontinuous galerkin methods for flow and transport in porous media. In *Challenges in Scientific Computing-CISC 2002*, pages 1–22. Springer, 2003.

[8] P. Bastian. A fully-coupled discontinuous galerkin method for two-phase flow in porous media with discontinuous capillary pressure. *Computational Geosciences*, 18(5):779–796, 2014.

 © by the authors, 2017

[9] S. Brdar, A. Dedner, and R. Klöfkorn. Compact and stable discontinuous galerkin methods for convection-diffusion problems. *SIAM Journal on Scientific Computing*, 34(1):A263–A282, 2012.

[10] R. Brooks and A. Corey. Hydraulic properties of porous media. *Hydrology Papers. Colorado State University*, (3), 1964.

[11] B. Cockburn and C.-W. Shu. The local discontinuous galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[12] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90(3-4):165–196, 2010.

[13] D. A. Di Pietro and A. Ern. *Mathematical aspects of discontinuous Galerkin methods*, volume 69. Springer Science & Business Media, 2011.

[14] J. Douglas and T. Dupont. Interior penalty procedures for elliptic and parabolic galerkin methods. In *Computing methods in applied sciences*, pages 207–216. Springer, 1976.

[15] Y. Epshteyn and B. Rivière. Fully implicit discontinuous finite element methods for two-phase flow. *Applied Numerical Mathematics*, 57(4):383–401, 2007.

[16] A. Ern, I. Mozolevski, and L. Schuh. Discontinuous galerkin approximation of two-phase flows in heterogeneous porous media with discontinuous capillary pressures. *Computer methods in applied mechanics and engineering*, 199(23):1491–1501, 2010.

[17] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[18] W. Klieber and B. Riviere. Adaptive simulations of two-phase flow by discontinuous galerkin methods. *Computer methods in applied mechanics and engineering*, 196(1):404–419, 2006.

[19] W. H. Reed and T. Hill. Triangularmesh methodsfor the neutrontransportequation. *Los Alamos Report LA-UR-73-479*, 1973.

[20] J.-F. Remacle, J. E. Flaherty, and M. S. Shephard. An adaptive discontinuous galerkin technique with an orthogonal basis applied to compressible flow problems. *SIAM review*, 45(1): 53–72, 2003.

[21] B. Riviere. Numerical study of a discontinuous galerkin method for incompressible two-phase flow. 2004.

[22] M. T. Van Genuchten. A closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil science society of America journal*, 44(5):892–898, 1980.

[23] M. F. Wheeler. An elliptic collocation-finite element method with interior penalties. *SIAM Journal on Numerical Analysis*, 15(1):152–161, 1978.

© by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 151–168
DOI: 10.11588/ans.2017.1.27447

System testing in scientific numerical software frameworks using the example of DUNE

Dominic Kempf[1] and Timo Koch[2]

[1]Interdisciplinary Center for Scientific Computing, Heidelberg University
[2]Institute for Modelling Hydraulic and Environmental Systems, University of Stuttgart

Received: January 31st, 2016; **final revision:** July 16th, 2016; **published:** March 6th, 2017.

Abstract: We present `dune-testtools`, a collection of tools for system testing in scientific software using the example of the Distributed Unified Numerics Environment (DUNE). Testing is acknowledged as indispensible support for scientific software development and assurance of software quality to produce trustworthy simulation results. Most of the time, testing in software frameworks developed at research facilities is restricted to either unit testing or simple benchmark programs. However, in a modern numerical software framework, the number of possible feature combinations constituting a program is vast. System testing, meaning testing within a possible end user environment also emulating variability, is necessary to assess software quality and reproducibility of numerical results. We provide an easy-to-use interface taking workload off developers and administrators in open-source scientific numerical software framework projects. In our approach, the large number of possible combinations is reduced using the scientific expert knowledge of developers to identify the practically relevant combinations. Our approach to system testing is designed to be integrated in the workflow of a research software developing scientist.

1 Introduction

Numerical software is nowadays an integral part of research at universities and other research facilities particularly considering natural sciences, engineering, and mathematics. Therefore, the quality of such software directly affects the effectiveness of research and the quality of research findings.

Often however, scientists develop and write research software themselves without advanced knowledge of or little experience in programming. The research may require to modify, develop, and understand the code in depth, ruling out using well-tested commercial software that may already exist. Also, time available for thoroughly testing software quality is limited. The situation may improve for researchers when joining their efforts to develop so-called numerical software frameworks. Frameworks provide features, i.e. software components, commonly used in the research area of interest. Thus, scientists profit from code reuse. What was already written by others does not have to be written again; what is already written can be further improved. Among the larger numerical software frameworks for solving partial differential equations, we mention

Dune [Bastian et al., 2008, 2011], FEniCS [Logg et al., 2012], and deal.II [Bangerth et al., 2016, 2007] as examples.

As many scientists rely on the same code, software quality becomes increasingly important in such software frameworks. Testing software quality is a vital process when developing and distributing software. Much theory and also practical guidance has been elaborated by the software engineering community. We mention [Myers et al., 2011] and [Burnstein, 2006] as well-written text books on software testing. Understandably, the focus lies on commercial software development models where testing is already an established process and software testers are suggested to be as many as developers themselves [Burnstein, 2006, p. 34]. In contrast, in numerical software framework projects, developing, testing, and research are usually conducted by the same persons. The developer is often focused on producing results rather than making the software reliable, maintainable, and usable for others. This means little time is available for testing. This paper focuses on quality measures and testing tools tailored for the numerical software community in universities and other research facilities.

In Section 2, the authors introduce quality measures important for numerical software frameworks. The challenges distinct to assuring quality in such frameworks are summarized. Section 3 introduces `dune-testtools` as the proposed approach to tackle the difficulties testing numerical software frameworks more thoroughly. The example of Dune as such a framework is introduced revealing the importance, difficulties, and the current lack of system testing. Section 4 and 5 explain the implementation and user interface of `dune-testtools` in detail. Read Section 6 to know how to get to our code. Finally, Section 7 gives an outlook to the authors' future concerns regarding the continuous integration of testing into the software framework development process at research facilities.

2 Quality and testing quality of numerical software frameworks

Numerical software frameworks differ vastly from software products considered in the software engineering community. One main difference, for example, is the small number of developers in numerical software frameworks that are also tester and scientist in personal union. Frameworks have by purpose a very large variability and combinability of features that makes them hard to test as a whole. We will therefore elaborate on our definition of software quality for numerical frameworks before proceeding with our approach.

2.1 Defining quality for numerical software frameworks

The IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990] provides a compact definition of software quality as "*[t]he degree to which a system, component, or process meets specified requirements . . . [and] user needs or expectations*". First, we therefore attempt to define typical quality characteristics expected from numerical software frameworks.

- Correctly implemented code: the code is expected to be implemented correctly with respect to syntax and functionality of features. It includes e.g. that the source code compiles with all compilers satisfying the communicated minimum requirements. Implementation correctness also includes runtime criteria like the absence of memory leaks and race conditions and proper error handling (e.g. through the use of exceptions).
- Numerically correct algorithms: the implemented algorithms should function as intended, namely reproduce results expected from theory. Often, there are well-known mathematical properties of an implemented algorithm that have to be satisfied.
- Trustworthy results: models and simulations built with the numerical software framework should produce trustworthy results, in the sense of correctly modeling what experiments or experts predict. Note that this is not solely a task for the framework but also for the scientist to use it as intended.

 © by the authors, 2017

- Good documentation, maintainability, testabilty: a high coverage of code documentation facilitates code reusage and also testing.
- Flexibility: in order to have many possibilities of combining features, the implementation and interfaces need to be designed with a certain level of abstraction. This is a key competence of numerical software frameworks.
- Sufficient performance and scalability.

2.2 Testing quality for numerical software frameworks

Testing is the process of determining how well these quality characteristics apply to a numerical software framework. In the scientific context, this process is often formalized in terms of verification and validation, e.g. [Oberkampf et al., 2002, Kelly and Sanders, 2008]. Hook and Kelly [2009] also consider what they call "code scrutinization", i.e. finding faults related to wrongly implemented code, among the testing processes. They further argue that in the scientific context it is usually not possible to determine whether a numerical software is correct because exhaustive testing is impossible due to the large number of possible program realizations. It is rather the goal to determine whether the software is trustworthy.

The process of code scrutinization directly corresponds to assuring correctly implemented code. Code scrutinization is a process early in the development phase and can be supported by complementing features with simple test programs. Verification of numerical code and its algorithms is the process of testing the algorithms for cases where the outcome is well-known. For example, for a simplified PDE and specific boundary and initial conditions there often exists an analytical solution that the numerical solution should converge to when refining the space and time discretization; it is also well-known that a Newton-Raphson scheme converges in one iteration for linear problems. In the examples, the numerical solution and the number of iterations are output parameters of a test and can be automatically checked against the reference values. Validation of a program or model is the process of analyzing the program output with expert knowledge or in comparison with experimental data. The scientist might know that in an experimental setup corresponding to the simulation setup the oil pumping rate lies between 4 and 5 kg/s. If the numerical model suits the experimental setup the code can be validated using the experimental parameter range. This can also be done automatically within a testing framework. The result should not change if the code makes use of different but also suitable discretization scheme or a different linear solver. Code scrutinization, verfication, and validation are processes suited for determining if the program produces trustworthy results. All three processes are outcome motivated, in the sense that they only evaluate results of a simulation program.

However, in the context of numerical frameworks, reusability, stability, and combinability play a key role assuring software quality. Testing variability and combinability can be viewed as the repetition of the above introduced processes for many possible realizations of programs and many possbile user configurations of those programs using features of a numerical software framework. Again, exhaustive testing is practically impossible.

2.3 Levels of testing

Burnstein [2006] decribes a need to categorize testing in four levels — unit testing, integration testing, system testing, and acceptance testing. Unit testing tests "*a single component*" [Burnstein, 2006, p. 133]. Unit testing is often available for numerical software frameworks and is good practice to write unit tests alongside every new feature.

Integration testing combines several components into working units. Integration testing can rarely be found in numerical software frameworks. If a certain interface is of great importance in a framework, sometimes interface checks are offered qualifying as integration testing. As the

modules in a modular framework are considered mostly independent, intermodular integration testing is often not sensible.

System testing, testing the combinations of features from several modules in a working numerical program, tests a fully working end user scenario in contrast to integration testing. System testing is not commonly employed in numerical software frameworks to the authors' knowledge. An exception are so-called tutorial, demo, or test programs testing specific simple setups with an educational intention. Sometimes framework include a suite of benchmarks which qualify as tests in the sense of verification and validation but generally do not test variability and feature combination. However, the vast combinatoric possibilities of a numerical framework require more comprehensive testing of module interdependencies in order to assure the reliablilty and generality of the framework. We will therefore have a focus on system testing subsequently.

Finally, acceptance testing is "*formal testing conducted to enable a user . . . to determine whether to accept a system or component*", according to the IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990]. In numerical software frameworks with an open-source development model, acceptance testing is constantly ongoing through the user community and feedback is commonly issued via mailing lists and issue trackers.

2.4 Regression testing

The code basis in open source numerical software frameworks is constantly under development and improvement. Those changes can be tracked by the testing suite by means of so-called regression testing. Burnstein defines that "*regression testing is . . . the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes.*" [Burnstein, 2006, p. 176]. For regression testing, the output of a numerical code is compared to a reference output created when the software was considered correct. Regression tests are possible on all four levels. Particularly, performing regression tests on a system test level has the ability to reveal if code changes might be an improvement in one configuration while being harmful for another.

2.5 System testing

System testing can be considered the most important testing level for a numerical framework, as the framework by design should allow arbitrary combination of modular blocks. In unit and integration testing, developers usually write tests covering their own code. In system testing however, code maintained by multiple developers is combined, revealing the bugs and corner cases the upstream developers have not thought of.

One approach to system testing within the software engineering community is to formalize software variability through Software Product Line Engineering (SPLE) [Pohl et al., 2005]. In SPLE, a *variability model* descibing thoroughly all possible configurations of the software is set up. Configurations usually vary in characteristic properties called *variation points*. Possible values of a variation point are called *variants*. A variability model for a complex software usually features a complex structure of constraints between variants.

Using SPLE in the context of a numerical framework for PDEs (DUNE) has already been investigated by Remmel et al. [2011] and Remmel [2014]. She states some of the fundamental inherent problems with SPLE for scientific frameworks: Determining a variability model for all the scientific applications is not possible due to the unlimited variability the mathematical input provides. Instead, SPLE is applied to model variability within the framework for a fixed mathematical model. This limited variability model and its constraints can only be determined with the knowledge of the developer.

 © by the authors, 2017

3 `dune-testtools` – A practical approach to integration and system testing in numerical software frameworks

This section will describe our approach to system testing of a numerical software framework. As our development is driven by the DUNE project as an example framework, we briefly introduce those properties of DUNE that influenced the design process.

DUNE, the Distributed Unified Numerics Environment, is an open source modular numerical software framework developed at more than 10 universities in Europe [Bastian et al., 2011]. It has a highly modular structure where currently five modules are *core modules* and as such mandatory for most applications. Built upon these core modules exist *discretization modules* like `dune-pdelab` (fast prototyping of new discretizations [Bastian et al., 2010]) and domain-specific application frameworks like DuMu^x (flow and transport processes in porous media [Flemisch et al., 2011]). Users will typically write their codes in form of *application modules* depending on the above. All those modules may further depend on external and third-party libraries. The modular structure of DUNE is managed by a CMake-based buildsystem. One of the design principles of DUNE is the definition of stable interfaces for common simulation components. The most famous example of this generic approach is the grid interface in `dune-grid` that allows to easily switch the underlying grid implementation of a code. Such a generic programming approach offers great opportunities for easily setting up system tests.

DUNE's open source workflow includes a group of core developers that have direct write access to the core repositories. Non-core developers contribute through merge requests. All users report issues through issue trackers and mailing lists. Most participants including the core developers develop DUNE next to their scientific research activity. As a consequence, the resources available for maintenance of the software are limited. Extensive testing has proven to suffer badly from this lack of resources in the past. To the authors' knowledge, the majority of DUNE developers work on Unix-like systems in a commandline-centered fashion and there is little to no usage of IDEs in the DUNE community.

From the above description of the DUNE project, the authors conclude the following requirements for a sustainable testing environment for modular software frameworks.

- Any solution for testing needs to integrate well into the existing modular structure of DUNE.
- To lower the entry barrier to a minimum, new workflows should integrate well with existing ones (no additional tools, IDEs etc.).
- Writing a new system test should be an easy and quick procedure.

As pointed out by Remmel et al. [2011] in the context of SPLE, the variability of framework-based scientific applications is twofold. On the one hand, there is the variability in the mathematical model serving as input data to the framework. On the other hand, given the output of the framework for a fixed mathematical model (an executable), there is variability in the configuration of the simulation run. See Figure 1 for an illustration.

The variability model for all scientific applications is too large to determine due to the infinite nature of the mathematical variation points. Only considering the variability model for a fixed mathematical input model also bears problems, as the variability model will still become very large. This results from the fact that the number of variation points for scientific software such as DUNE is high and the variability model is the combinatoric product of these variants. Therefore, a strategy to reduce the amount of test cases for system testing while assuring sufficiently high coverage is needed.

We base our approach on the idea, that the entire DUNE user base needs to be involved in the quality assurance process. This is important for the following reasons.

© by the authors, 2017

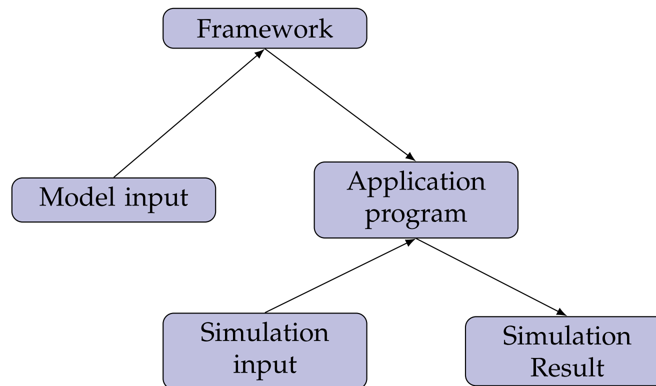


Figure 1: A simulation is set up in two stages. Firstly, a mathematical model is selected and the necessary features for implementing this model are chosen from the features offered by the numerical framework. Secondly, the application program is configured by the simulation input being compile time or runtime parameters. Both stages are objected to a large variability.

- The entirety of system tests should cover all the application areas of DUNE. Those are however defined by the users putting efforts into applying DUNE to new areas. In other words, the only sensible sampling of the variability model that includes the mathematical models is to use those samples that DUNE users are actively working on.
- A system test is best written by the expert in the field, who is the user working on the exact problem. Also, defining a reasonably large sample of the variability model for a given model is only possible with good scientific knowledge of the problem. For instance, the choice of linear solver and/or preconditioner for a PDE problem cannot be done a priori.
- Reusing user code for system tests can result in a mutual benefit for user and framework. On the one hand, the user contributes samples to the framework's test suite that increase the coverage of the variability model. On the other hand, the user gets an automated testing workflow for his or her code, increasing the quality and reproducibility of the code and the scientific work and results.

To sum it up: instead of providing a full variability model, we carefully choose a set of system tests that try to cover as many use cases of the DUNE project as possible. For a given system test, we again do not provide the full variability of the framework, but instead go for samples carefully selected by the user. These samples should cover all of the configurations relevant to the problem. The system tests in their entirety should aim for a full coverage of relevant variant combinations of the framework.

Considering tools for test evaluation, we heavily rely on regression testing. In a steadily changing code base, it is of particular importance to track regressions in a system testing context. We store reference files, created when the code was considered correct, i.e. usually around the time of a new version release, under version control. We mention the recently introduced large file storage concept of Git [Git-lfs webpage] as solution to the problem of storing large files under version control. Keeping reference files over a longer period of time eliminates the chance for slow regression and reduces the testing effort in comparison to using ad-hoc before and after comparisons whenever the code base is changed.

We provide an implementation of the above presented concepts in the form of the DUNE module `dune-testtools`. Having the implementation as a DUNE module integrates it easily into the modular structure of the DUNE project. `dune-testtools` only provides means to *define* system tests independently of the system the test will be run on. `dune-testtools` contains a Python package `dune.testtools` which does the heavy lifting of the implementation. Furthermore,

 © by the authors, 2017

it features CMake code for the buildsystem integration and some C++ helper classes. The following Sections 4 and 5 elaborate on how `dune-testtools` implements feature modeling and test evaluation, respectively.

4 Feature modeling in `dune-testtools`

As mentioned previously, our approach relies heavily on the application developers to define the variability model for their use cases. To reduce entry barriers to writing system tests to a minimum, we try to integrate our approach as deeply as possible into the existing workflow of DUNE developers. As a consequence, we completely avoid external tools, but instead work with what we identified as the integral parts of a simulation code:

- A C++ source file that runs the simulation,
- an ini file (i.e. a parameter file) that allows to configure the simulation run, and
- little CMake code to define executable and tests.

The C++ source file is only to a limited extent suitable for modelling variability. Code will get duplicated too often. It will look cluttered and as a consequence it will not be maintainable. However, we consider maintainability one of the crucial factors for the success of the quality assurance process. We therefore decided to use the ini file for modelling variability by extending its syntax to what we call *meta ini files*. Instead of just one configuration, meta ini files describe a set of combinations and therefore serve as a domain specific language for modelling the variability model for our system tests. Meta ini files are expanded into a set of normal ini files in a preprocessing step.

4.1 Extending ini files

Normal DUNE-style ini files contain key/value pairs, which are separated by an equal sign (=). Keys can be grouped by sections which may either be defined explicitly by bracing its name with square brackets, or implicitly by using dots in key names. Sections may be nested to arbitrary depth.

In meta ini files, we introduce commands which can be applied to key/value pairs with a Unix-style pipe (|). The most important command is the `expand` command, as it provides the mechanism to describe sets of configurations. The values of keys that have the `expand` command applied are expected to be comma-separated lists. Those lists are split and the list of configurations is updated to hold the product of all possible values. The following example yields a total of 6 configurations describing all the combinations of values for `theta` and `timestep`.

Meta ini Code

```
1  theta = 0.5, 1.0 | expand
2  timestep = 0.1, 1.0, 10.0 | expand
```

Sometimes, you may not want to generate the product of possible values, but instead couple multiple key expansions. You can do that by providing an argument to the `expand` command. All `expand` commands with the same argument, will be expanded together. The expansion process when having `expand` commands with the same argument but a differing number of comma separated values is semantically not well-defined and therefore produces an error. The next example yields 2 configurations and does a refinement in space and time simultaneously.

Meta ini Code

```
1  gridlevel = 1, 2 | expand refinement
2  timestep = 1.0, 0.5 | expand refinement
```

© by the authors, 2017

Expansion with and without argument may be combined arbitrarily. The following example combines the two above examples for a total of 4 configurations.

Meta ini Code

```
1  theta = 0.5, 1.0 | expand
2  gridlevel = 1, 2 | expand refinement
3  timestep = 1.0, 0.5 | expand refinement
```

Sometimes, you will want to define a key's value depending on the value of another key, where that other key has the `expand` command applied. You may do so by writing the keyname in curly brackets into the values. See the following example, where an output naming scheme is controlled through this mechanism.

Meta ini Code

```
1  dimension = 2, 3 | expand
2  boundarycondition = dirichlet, neumann | expand
3  outputfile = {boundarycondition}_{dimension}d.output
```

The curly bracket syntax may even be used in a nested way and provides a powerful tool for the meta ini language.

With the `expand` command as shown above, a set of configurations needs to have a product structure in order to be described by a meta ini file. To overcome this limitation, the `exclude` command has been introduced. It allows to define a boolean condition using curly brackets. If that condition evaluates to `True` in Python, the entire configuration is removed from the set of configurations. Because the `exclude` command does not operate on a key/value pair by definition, it is instead applied to the condition directly in a separate line of the ini file. In the following example, the grid levels that are not suitable for three-dimensional simulations are discarded. It yields a total of 8 configurations, 5 of which are for the two-dimensional and 3 of which are for the three-dimensional case.

Meta ini Code

```
1  dimension = 2, 3 | expand
2  gridlevel = 1, 2, 3, 4, 5 | expand
3  {dimension} == 3 and {gridlevel} > 3 | exclude
```

Once the set of generated configurations starts to grow, it is helpful to control the naming scheme of the generated ini files. We use the special key `__name` for that. You may define it according to your needs using the curly bracket syntax. If your definition does not result in unique names, consecutive numbering will be appended to your naming scheme. The `__name` key is optional and not defining it will result in consecutively numbered ini files.

So far, we have only considered the `expand` and the `exclude` command. There are some more commands implemented which we only mention briefly here and refer to the documentation of `dune-testtools` for further reading.

- `unique` makes the values to the given key unique throughout the set of configurations, useful for unique names like output filenames. The `__name` mechanism uses this command internally.
- `eval` allows to perform simple arithmetic operations on keys
- `tolower` and `toupper` transform strings to their lower-, uppercase representations, respectively.

Note that the meta ini syntax uses some characters for defining its own syntax. These are

- `{` and `}` in values for key-dependent resolution,

 © by the authors, 2017

- | in values for piping commands,
- , in comma separated value lists when using the `expand` command.

If you want to use those characters as part of a value, you need to escape either by a preceding backslash or by embracing quotes.

4.2 Static and dynamic variations

So far, we have introduced meta ini files as a way of expressing the variability model for simulation codes. Each variation point is represented by a key which has the `expand` command applied. However, one of the biggest challenges with the variability model of simulation environments such as DUNE is the fact that many common variation points deal with compile-time variants. Unfortunately, not all variation points describe static variations and treating dynamic variations as static ones will result in a waste of resources. We will therefore describe in this section, how to extend the concept of meta ini files by the ability to describe static and dynamic variations in a mixed fashion.

We introduce a special group in the meta ini file called `__static`. Any key/value pairs in the `__static` section are to be passed as C preprocessor variables to the simulation executable. To that end, the expansion process and the buildsystem need to be interleaved. We will not cover the details of this interface from CMake to Python here. Instead, we describe the CMake API from `dune-testtools` in detail in Section 4.3. Here, we restrict ourselves to describe how to add static variations to meta ini files. See the following minimal example that defines the grid type as a variation point.

Meta ini Code

```
1 [__static]
2 GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand
```

When used with the macros described in Section 4.3, CMake will parse this meta ini file at configure time and generate executables from the static section. In analogy to the `__name` mechanism, you can control the naming scheme of the executables by defining the key `__exec_suffix` with the curly bracket syntax. The suffix will be made unique and be appended to the target basename given to CMake.

To correctly set up executables and tests for a system test with both static and dynamic variations, CMake needs to determine the correct number of static variants and the correct number of dynamic variations per executable. While the first one is easily done by filtering the `__static` section from the set of configurations, the latter bears some problems. We implicitly implied until now that any non-`__static` key defined a dynamic variation. However, when writing more sophisticated meta ini files, you might want to define some auxiliary keys not defining dynamic variations. Take the following advanced example where auxiliary keys are used to define a polynomial degree depending on the grid.

Meta ini Code

```
1 grid = yasp, ug | expand grid
2 deg_yasp = 2
3 deg_ug = 2, 3 | expand
4 [__static]
5 GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
6 DEGREE = {deg_{grid}}
```

The example would produce a total of four test from two executables. Two of these tests would effectively be equivalent as they are using `Dune::YaspGrid<2>` as their grid type and only differ in the unused value `deg_ug`. To overcome this limitation, we have to mark the keys `deg_yasp` and `deg_ug` as auxiliary keys by putting them into the `__local` group. The following corrected example produces the desired amount of three tests.

© by the authors, 2017

Meta ini Code

```
grid = yasp, ug | expand grid
[__local]
deg_yasp = 2
deg_ug = 2, 3 | expand
[__static]
GRIDTYPE = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
DEGREE = {__local.deg_{grid}}
```

The keys in the `__local` group are expanded and they are available for curly bracket syntax resolution. However, at the end of the expansion process, the `__local` section is removed from the generated configurations and the set of configurations is shrunk by eliminating duplicate configurations.

Using the meta ini mechanism to have CMake automatically add targets bears another problem if you are used to guard the addition of executables in CMake with `if`-clauses that depend on configure checks for external packages. Consider the above example in a situation, where the external dependency UG was not found on the system. In that case, the executable should not even be added in the buildsystem. Such guards need to be defined in the meta ini file through the command `cmake_guard`, which similar to the `exclude` command does not operate on a key/value pair but on a single value. This value is evaluated in CMake at configure time and the executable is not added, if it evaluated to `FALSE`. The above example can be fixed as follows. `TRUE` is used as a non-operational guard variable here, as `Dune::YaspGrid` is always available.

Meta ini Code

```
[__static]
GRID = Dune::YaspGrid<2>, Dune::UGGrid<2> | expand grid
TRUE, HAVE_UG | expand grid | cmake_guard
```

CMake provides a very useful mechanism to attach metadata to tests in the form of so called labels. Tests can have an arbitrary amount of labels attached and labels do not have any semantics by themselves. The testing driver `ctest` can be used with regular expressions on test labels. Being able to define labels in meta ini files allows us to define different priority groups. This can be done through the command `label` which, as `exclude`, operates on a Python Boolean expression. It takes the label to be applied as the first argument. Similar to the `expand` command, you may provide an arbitrary identifier as the second parameter. Labels with the same identifier will override previously defined labels with the same identifier. Consider the following example where all tests have the `WEEKLY` label applied, but one labeled `NIGHTLY`. Note how `True` is used to set the default label `WEEKLY` for all configurations.

Meta ini Code

```
model = 1, 2, 3, 4, 5 | expand
True | label WEEKLY PRIORITY
{model} == 3 | label NIGHTLY PRIORITY
```

Writing system tests with `dune-testtools`, you might experience that a generic coding style within the C++ source file is very beneficial. You will often define types for simulation components in the meta ini file. If all possible variants for a given static variation point fulfill the same interface or concept, you can do that with a single codepath and thus achieve maintainability. We have identified this need and try to provide tools for the most common tasks either through `dune-testtools` or through contribution to the DUNE core modules. This currently includes

- the `IniGridFactory` from `dune-testtools`, which allows to create grids from a section in an ini file. It provides specializations for the most important DUNE grids.
- The ongoing project [dune-istl GitLab] of providing fully dynamic, ini file-based solver and preconditioner setup in `dune-istl`.

 © by the authors, 2017

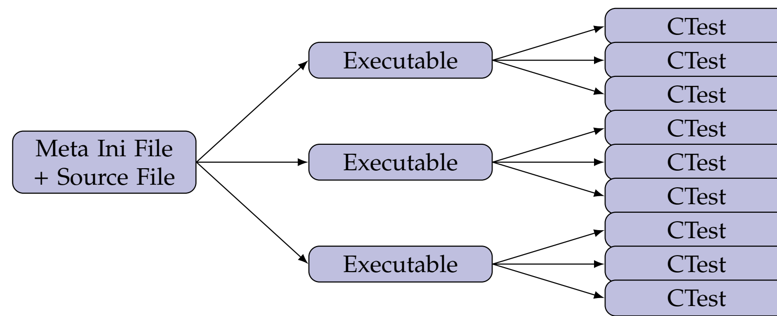


Figure 2: Expansion procedure for the macro `dune_add_system_test`

4.3 Incorporation into the build system

One of the core principles of `dune-testtools` is to provide a powerful buildsystem API to the user, not requiring any specific CMake knowledge on the user side. The interface to add a system test with both static and dynamic variations is the following CMake macro.

CMake Code

```
dune_add_system_test(SOURCE src
                     BASENAME base
                     INIFILE ini
                     TARGETS output
                     [SCRIPT script])
```

Writing such a block into a `CMakeLists.txt` file will trigger the following procedure, which is illustrated by Figure 2.

- The given `INIFILE` will be expanded into the current build directory.
- CMake will add executables that are built from the given `SOURCE` with a naming scheme based on the given `BASENAME` with appended `__exec_suffix`. A list of the generated executables is exported to the variable given to `TARGETS`. You may use this to alter targets by adding flags etc. We recommend using the `dune_enable_all_packages` feature from `dune-common` though, to minimize problems arising from the necessity to add specific flags to the generated executables.
- Tests will be added to the `ctest` testing suite for all dynamic variants. The executables are expected to take an ini file as their first and only command line argument.

The `SCRIPT` parameter will be explained in detail later on, as it allows you to attach testing tools to your system tests.

`dune-testtools` also provides CMake macros to use meta ini files outside of the context of testing. These can be useful for setting up reproducible numerical studies. We do not describe these here, but instead refer to the buildsystem documentation of `dune-testtools`.

5 Verification and validation with `dune-testtools`

A test of a numerical code will usually run a small sample problem and return some data. The test suite then needs to determine whether the test passed or failed. The criterium for success of a test is directly related to quality measures the test is supposed to verify. To avoid rewriting the testing code in all tests' executables, `dune-testtools` implements testing methods as wrapper scripts. Wrapper scripts in `dune-testtools` are Python scripts controlling pre-processing, execution of

© by the authors, 2017

the program, and post-processing. The wrapper scripts decide whether a test passed or not. It supplies useful information to the developer in case it did not. The wrapper scripts are meant to be customized writing few lines of Python code, although `dune-testtools` already provides many useful wrappers. The wrappers are configured by the meta ini file in the section `wrapper.<wrapperscriptname>`.

The wrapper script for a system test is defined through the above introduced CMake macro `dune_add_system_test` by setting the SCRIPT parameter. As of the writing of this article, `dune-testtools` contained the wrapper scripts `dune_execute.py`, `dune_execute_parallel.py`, `dune_outputtreecompare.py`, `dune_vtkcompare.py`, `dune_convergencetest.py`. These wrappers are described in more detail in this section.

5.1 Simple execution wrappers

The default wrapper `dune_execute.py` is used whenever the SCRIPT parameter is omitted in CMake. It runs the executable and forwards the return code.

The wrapper `dune_execute_parallel.py` runs the executable in parallel and forwards the return code. The wrapper would be defined in the CMake macro as follows.

CMake Code

```
dune_add_system_test(...
                     SCRIPT dune_execute_parallel.py)
```

The number of processors for the parallel run can then be specified in the meta ini file.

Meta ini Code

```
[wrapper.execute_parallel]
numprocessors = 4
```

The MPI implementation is automatically detected by the build system and information on it is passed to the wrapper script.

5.2 Regression tests

Assume we have a reference program output produced when the state of the program was considered correct. Regression testing compares the output of the software's current state to this reference output, as introduced in Section 2.4. A very common output of numerical simulation is a computational result in a VTU format [VTK] file, e.g. a discrete solution on an unstructured grid. `dune-testtools` provides the wrapper script `dune_vtkcompare.py` for regression testing VTU output.

Technical difficulties comparing such unstructured grid files include that the data is commonly given in single precision floating point numbers which need to be compared with a certain tolerance value (*fuzzy comparison*). A further problem inherent to floating point numbers is the comparison of numbers that are physically zero but suffer from numerical noise. Finally, different grid managers might use different ordering of vertices or cells.

The wrapper script `dune_vtkcompare.py` offers solutions to tackle these problems. To this end, before comparing the data, both files are sorted by ascending vertex coordinates. Then, different sets of parameters are sorted by their name attribute, because the parameter order in VTU files is mutable. Floating point numbers are first compared by means of an absolute comparison, i.e. two single precision floating point numbers a and b are considered equal if the absolute value of their difference is smaller than the machine epsilon scaled with the magnitude of the parameter, i.e. if $|a-b| <= \epsilon \cdot \max(\mathcal{P})$. Herein, $\mathcal{P}$ denotes the set of absolute parameter values given in the VTU file. The machine epsilon for IEEE 754 single precision floating point numbers is $\epsilon \approx 1.19 \cdot 10^{-7}$, according to the definition of Goldberg [1991].

 © by the authors, 2017

If a and b are not equal, the script relaxes the criterium and compares relatively. The two floating point numbers are considered being close enough if $|a - b| <= \delta \cdot \max(|a|, |b|)$, where δ is the threshold for the largest acceptable deviation. The relative criterium is necessary as a numerical code introduces rounding errors for every floating point computation. The errors can easily sum up to relatively large errors.

If a data set is physically zero, e.g. the z-component of the velocity in a simulation was fixed to zero but the actual number is subjected to computations with rounding errors, the result can be an array of numbers very close to zero considered as numerical noise. In a case like that, two numbers $v_{z,1} = 1.5 \cdot 10^{-21}$ and $v_{z,1} = 1.2 \cdot 10^{-18}$ are considered zero but will fail both the absolute and relative criterium. For these cases the scripts provides a zero threshold per parameter under which a parameter will be considered zero and thus be exempted from comparison.

Following the above mentioned concepts, the wrapper script `dune_vtkcompare.py` can be configured via the meta ini file as follows.

Meta ini Code

```
1 [wrapper.vtkcompare]
2 name = my_vtkfile
3 reference = path_to_reference_file
4 extension = vtu
5 relative = 1e-2
6 absolute = 1.2e-7 # set to machine epsilon of used floating point format
7 zeroThreshold.velocity = 1e-18
```

`name` and `reference` are mandatory parameters denoting the file to be compared and the reference file, respectively. The path `name` is relative to the build directory of the test and the path `reference` is relative to the source directory. `extension` specifies the file extensions of the files to be compared and defaults to `vtu`. The parameters `relative` and `absolute` define the thresholds for relative and absolute floating point comparison. Note that the absolute threshold will be scaled to the magnitude of the data to compare. They default to $\delta = 10^{-2}$ and $\epsilon = 1.2 \cdot 10^{-7}$, respectively. The zero threshold can be set for every parameter in the VTU file by name (`velocity` in the above example).

The wrapper script `dune_outputtreecompare.py` does regression testing with arbitrary user data, where the ini file format is used for the output. `dune-testtools` provides a C++ class `Dune::OutputTree` that facilitates writing data to ini files. It inherits from `dune-common`'s `Dune::ParameterTree` and adds some convenience methods for writing out data. The following code shows how to instantiate a `Dune::OutputTree` and fill it with data.

C++ code

```
1 Dune::OutputTree outputTree("out.ini");
2 // execute code yielding a variable numIterations
3 outputTree.set("Iterations", numIterations);
```

Upon destruction, the contents of the `Dune::OutputTree` will be automatically dumped to a file. The wrapper script comparing ini files can be configured as follows via the test's meta ini file.

Meta ini Code

```
1 [wrapper.outputtreecompare]
2 name = myoutputfile
3 reference = path_to_reference_file
4 extension = out
5 type = fuzzy
6 exclude = DebugInfo
7 relative = 1e-2
8 absolute = 1.5e-7
9 zeroThreshold.norm = 1e-18
```

© by the authors, 2017

Most options work as explained above for the VTU file comparison. The parameter `type` specifies whether the files are compared `exact` (not recommended when floating point numbers are involved) or a `fuzzy` comparison is preferred. When comparing fuzzy, every value in the ini file that is convertible to a floating point number will be compared as such using the absolute and relative thresholds specified. A key only written for e.g. debugging purposes that is not required to equal the reference file can be excluded by specifying the `exclude` key that takes a space separated list as value. Multiple such output data can be compared in a single test. For configuring the comparison differently for multiple files the following scheme applies.

Meta ini Code

```
[wrapper.outputtreecompare]
name = myoutputfile1 myoutputfile2
reference = path_to_reference_file1 path_to_reference_file2
extension = out out

[wrapper.outputtreecompare.myoutputfile1]
type = exact
exclude = DebugInfo

[wrapper.outputtreecompare.myoutputfile2]
type = fuzzy
zeroThreshold.norm = 1e-18
```

5.3 More involved test wrappers

Several quality measures of numerical software can only be tested with more involved tests often requiring several runs of the same executable. A before mentioned method to verify an algorithm for e.g. a discretization scheme solving a PDE is a convergence study. The grid is refined several times and a rate of convergence can be computed. The rate is inherent to the scheme and should not change when the code base changes. `dune_convergencetest.py` implements a test wrapper for convergence tests. It does require the program to compute an error, e.g. with respect to a known analytical solution. The key in the meta ini file that defines the samples for the convergence test has to marked with the `convergencetest` command, which is a special form of the `expand` command.

Meta ini Code

```
[grid]
refinement = 0, 1, 2, 3, 4 | convergencetest
```

The above example defines a test conducting a grid convergence study. The convergence test wrapper can be configured via the same meta ini file.

Meta ini Code

```
[wrapper.convergencetest]
expectedrate = 2.0
absolutedifference = 0.1
```

The test runs the executable once for every level of refinement. It then computes the convergence rate automatically and compares it to the expected one. In the above example the test would fail if the computed rate differs by more than 0.1 from 2. To compute the rates correctly we make some assumptions on how the data is dumped by the program. The following dummy code snippet taken from `dune-testtools` outlines such a test using DUNE.

C++ code

```
#include <dune/testtools/outputtree.hh>
#include <dune/common/parametertreeparser.hh>
#include <dune/common/parametertree.hh>

int main(int argc, char** argv)
```

 © by the authors, 2017

```
6  {
7      // read the given ini file
8      Dune::ParameterTree params;
9      Dune::ParameterTreeParser::readINITree(argv[1], params);
10
11     // get refinement level
12     auto refinement = params.get<int>("grid.refinement");
13     // ... and refine the grid accordingly
14
15     // construct an output tree with the ini tree
16     Dune::OutputTree outputTree(params);
17
18     // do simulation and compute an error norm and hmax
19
20     // output convergence data
21     outputTree.setConvergenceData(norm, hmax);
22     return 0;
23 }
```

Lines 7-9 read the expanded ini file associated with the run into a `Dune::ParameterTree` object. A `Dune::OutputTree` can be constructed from the input ini file to avoid specifying its filename redundantly. Line 21 shows how to use a convenience method of the `Dune::OutputTree` class to easily dump a computed error norm and the global maximum grid element diameter to file.

Other more involved testing methods are e.g. scalability tests where a program is run with different numbers of processors and a speed-up per additional processor can be observed. Performance tests measure the runtime of a certain program. Code changes should usually not increase runtime (this might be a necessary evil of another advantage), whereas a decreased runtime is considered an improvement. The latter tests are architecture dependent and can only be executed in comparison to a reference state obtained on the same machine. A possible scenario would be to run such tests before and after a change to the code base. Such tests can be automated when using appropriate version control systems. Scalability and performance testing will appear in future releases of `dune-testtools`.

5.4 Writing new test wrappers

The existing test wrappers can be customized easily with little knowledge of Python. For example, the parallel wrapper could be combined with the wrapper comparing output data, to test if the output from a parallel program run is equal to that of a sequential execution. If you want to write a wrapper from scratch, check the Python module `dune.testtools.wrapper.argumentparser` for a list of information that CMake forwards to the wrapper scripts. Following `dune-testtools`'s style of configuring test wrappers avoids confusion.

6 Code Availability and Portability

The code presented in this paper is available freely under a BSD license. You find an overview of the quality assurance related projects of the authors under

https://gitlab.dune-project.org/groups/quality

Note, that `dune-testtools` also requires the module `dune-python`, which is also available under the same URL. `dune-python` is a buildsystem extension to `dune-common` that unifies the ways of distribution and installation of DUNE modules and Python packages. The centerpiece of `dune-python` is a Python virtualenv, a tool setting up a system-independent environment for running Python code. It is automatically set up at configure time in the build directory. `dune-python` is designed to be used by any other DUNE module distributing Python code.

As can be expected from a project on quality assurance, we have taken measures to ensure the quality of our code. Extensive unit testing is done for Python and C++ code and as well for

CMake code through configure time unit tests. We do provide comprehensive documentation, which can be built by typing `make doc` in the toplevel build directory. In `dune-testtools`, you find the result in the (build directory) subdirectory `doc/sphinx/html`.

The methods presented in this paper are - though tailored for it - not limited to usage with the DUNE project. The Python package `dune.testtools`, which contains the code for both meta ini expansion and testing tools is completely general and does not contain any DUNE specifics. The interface between the Python package and the projects buildsystem, however, needs to be rewritten in order to use it with a different numerical code.

7 Outlook – an open-source workflow for automated system testing in numerical software frameworks developed at research facilities

The module `dune-testtools` presented in this paper is only a first step towards an automated system testing workflow. It provides convenient, buildsystem integrated tools to define low dimensional samples of the high dimensional variability model of simulation codes based on DUNE. Furthermore, it provides tools to verify and validate results for the numerical solution of partial differential equations specifically. All these tools focus on the definition of tests independent of the environment that the tests are to be run in. The system tests defined with `dune-testtools` may be run on any local machine by running `ctest` in the build directory.

In future work we will target the infrastructure for automating the execution of system tests. This includes the following tasks:

- A buildbot [Buildbot webpage] based build server setup
- Using docker [Docker webpage] containers to model heterogeneous userlands
- Integrating buildbot into a merge-request based development model

8 Acknowledgments

The authors would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart, as well as the Ministry of Science, Research and the Arts, Baden-Württemberg, Germany.

References

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. doi: 10.1145/1268776.1268779. URL http://dx.doi.org/10.1145/1268776.1268779.

W. Bangerth, D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. The deal.II library, version 8.4. *Journal of Numerical Mathematics*, 24, 2016. doi: 10.1515/jnma-2016-1045. URL http://dx.doi.org/10.1515/jnma-2016-1045.

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, 2008. doi: 10.1007/s00607-008-0003-x. 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(2):294–315, 2010. URL http://www.kybernetika.cz/content/2010/2/294/paper.pdf.

 © by the authors, 2017

P. Bastian, M. Blatt, A. Dedner, C. Engwer, J. Fahlke, C. Gersbacher, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, S. Müthing, M. Nolte, M. Ohlberger, and O. Sander. DUNE Webpage, 2011. http://www.dune-project.org.

Buildbot webpage. http://buildbot.net.

I. Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.

Docker webpage. https://www.docker.com/.

dune-istl GitLab. The project is developed on the branch `feature/istl-ini` in the `dune-istl` repository. See the merge request https://gitlab.dune-project.org/core/dune-istl/merge_requests/5 for an overview what is happening. [Accessed: February 2017].

B. Flemisch, M. Darcis, K. Erbertseder, B. Faigle, A. Lauser, K. Mosthaf, S. Müthing, P. Nuske, A. Tatomir, M. Wolff, et al. DuMu^x: Dune for multi-{phase, component, scale, physics,...} flow and transport in porous media. *Advances in Water Resources*, 34(9):1102–1112, 2011. doi: 10.1016/j.advwatres.2011.03.007. URL http://dx.doi.org/10.1016/j.advwatres.2011.03.007.

Git-lfs webpage. https://git-lfs.github.com/.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991. doi: 10.1145/103162.103163. URL http://dx.doi.org/10.1145/103162.103163.

D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 59–64, May 2009. doi: 10.1109/SECSE.2009.5069163. URL http://dx.doi.org/10.1109/SECSE.2009.5069163.

IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064. URL http://dx.doi.org/10.1109/IEEESTD.1990.101064.

D. Kelly and R. Sanders. Assessing the quality of scientific software. In *First International Workshop on Software Engineering for Computational Science and Engineering*, 2008. URL http://secse08.cs.ua.edu/Papers/Kelly.pdf.

A. Logg, K.-A. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8. URL http://dx.doi.org/10.1007/978-3-642-23099-8.

G.J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

W. Oberkampf, T. Trucano, and C. Hirsch. Verification, validation and predictive capability in computational engineering and physics. In *Hopkins University*, pages 345–384, 2002. doi: 10.1115/1.1767847. URL http://dx.doi.org/10.1115/1.1767847.

K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. ISBN 978-3-540-24372-4. doi: 10.1007/3-540-28901-1. URL http://dx.doi.org/10.1007/3-540-28901-1.

H. Remmel. *Supporting the Quality Assurance of a Scientific Framework*. PhD thesis, 2014.

H. Remmel, B. Paech, C. Engwer, and P. Bastian. Supporting the Testing of Scientific Frameworks with Software Product Line Engineering: A Proposed Approach. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, pages 10–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0598-3. doi: 10.1145/1985782.1985785. URL http://doi.acm.org/10.1145/1985782.1985785.

© by the authors, 2017

VTK. The Visualization Toolkit: File Formats. http://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf [Accessed: February 2017].

 © by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 169–192
DOI: 10.11588/ans.2017.1.27477

FunG - Automatic differentiation for invariant-based modeling

Lars Lubkoll

Received: January 31st, 2016; **final revision:** July 9th, 2016; **published:** March 6th, 2017.

Abstract: This document describes a `C++` -library for the generation of invariant-based models, including first three derivatives. Using expression templates, features of `C++11/14`, forward automatic differentiation and modern SFINAE-techniques admits a highly efficient implementation with a simple and intuitive interface.

1 Introduction

Efficient and robust mathematical algorithms for the solution of real-world and scientific problems often require the computation of derivatives of complex functions. Since the manual computation of derivatives is time-consuming and error-prone, this problem is typically solved by software. Different strategies have been developed to solve this task. The most popular approach, finite differences, is known to cause significant problems due to truncation errors and the difficulty to control their accuracy for highly nonlinear problems.

For this reason several other approaches have been investigated and implemented. Besides finite differences, the most popular strategies for the computation of derivatives are symbolic and automatic differentiation (AD). The former computes the derivative of a function, which then can be evaluated, whereas the latter directly computes the evaluation of the derivative.

In `C++` the standard implementation technique for both symbolic and automatic differentiation is to provide overloads for arithmetic operators and elementary functions in `cmath`, resp. in `math.h` such that their direct evaluation can be replaced by an object that carries all information necessary for the computation of derivatives. One distinguishes forward- or reverse-mode schemes. Forward-mode schemes implement the same procedure that one typically uses for the manual computation of derivatives. Starting from the whole function, differentiation rules are applied recursively until we reach the point where only simple expressions, for which the derivatives are known, must be processed. Reverse-mode schemes use a more involved strategy for the evaluation of derivatives and are not of interest here[1]. The interested reader is referred to eg. Griewank and Walther [2008].

An incomplete list of forward-mode schemes includes ADOLC Walther and Griewank [2012], though with main focus on reverse-mode schemes, Sacado Heroux et al. [2005], which is part of the Trilinos project, CoDiPack Sagebaum et al., a particularly efficient implementation of the scientific computing group of Technische Universität Kaiserslautern, ADEPT Hogan [2014],

[1] For the use-cases that are of interest here, reverse-mode schemes are significantly slower than forward schemes.

FADBAD++ Bendtsen and Stauning [1996] and CppAD Bell. Other approaches use symbolic differentiation to compute the derivative of a function. This strategy is followed in SEMT SEM, Gil and Gutterman [1998] and in Kourounis et al. In particular the latter seems to be interesting. Unfortunately the article was never published and the used code is not publicly available[2]. For further references the interested reader is referred to Kourounis et al. and the website www.autodiff.org.

Despite the large number of available AD-libraries FunG[3] provides some unique features. First it is not tied to a specific type of variable. Instead it provides a small and flexible interface that admits to work with general vector space elements. In particular FunG admits the computation of directional derivatives with respect to matrices and provides all functionality for invariant-based models, i.e. models that describe physical processes in terms of matrix invariants. This type of model is the basis of state of the art models for hyperelastic materials, cf. Lubkoll [2015] and the references therein. For these materials the mechanical properties are given in terms of a strain density function $W(F)$ that depends on the deformation gradient $F \in \mathbb{R}^{3,3}$. As an example we consider a model for the description of muscle tissue from Martins et al. [1998], given through

$$
\begin{aligned}
W(F) &= c\left[\exp(b(\bar{\iota}_1(C) - 3)) - 1\right] + d\left[\exp(a(\bar{\iota}_6(C,M) - 1)^2) - 1\right], \\
\bar{\iota}_1(C) &= \operatorname{tr}(C)\det(C)^{-1/3}, \qquad C = F^T F, \\
\bar{\iota}_6(C,M) &= \operatorname{tr}(CM^2)\det(C)^{-1/3},
\end{aligned}
$$

where $\bar{\iota}_1$ is called the first modified invariant and $\bar{\iota}_6$ the third modified mixed invariant. The latter is used to describe the anisotropic influence of the muscle fibers. The need to assemble $\int_\Omega W^{(n)}(F)\,dx,\ n = 1,\ldots,3$ was the motivation for the development of FunG.

Another advantage is the runtime performance of the generated models, which is demonstrated in Sec. 5. This has several reasons. The treatment of multiple variables is different from other implementations and admits to avoid redundant computations. Furthermore, due to a compiler-friendly code design, the generation of efficient code is strongly simplified. Careful profiling and suitable optimizations yield an implementation that contains only small runtime overhead.

Eventually, all other investigated AD-implementations require the introduction of additional types. Mostly these are containers for scalar unknowns. Only FADBAD++ does not need these wrappers, though it requires to use a particular container for the implementation of the function for which we want to compute the unknown. In this respect FunG requires less setup code for using it. For the above introduced model of muscle tissue we can simply write:

C++ code

```
1  using namespace FunG;
2  // for given type Matrix
3  // initialize the structural tensor M
4  auto M = LinearAlgebra::unitMatrix<Matrix>();
5  // initialize the deformation gradient F
6  auto F = LinearAlgebra::unitMatrix<Matrix>();
7  // generate relationship between strain and deformation gradient
8  auto C = LinearAlgebra::strainTensor(F);
9  // generate the relationship between strain and energy density
10 auto W = c*( exp( b*( mi1(C) - 3 ) ) - 1 )
11           + d*( exp( a*pow<2>( mi6(C,M) - 1 ) ) - 1 );
```

Here `mi1` and `mi6` denote the first modified invariant $\bar{\iota}_1(F)$, resp. the third modified mixed invariant $\bar{\iota}_6(F,M)$ and the function `strainTensor(F)` generates the strain tensor $C = F^T F$.

[2] A preprint of the paper is available at www.researchgate.net/profile/Michael_Saunders2/publications.

[3] The name FunG is both an abbreviation for Function Generation and a tribute to Yuan-Cheng Fung one of the founders of modern biomechanics.

 © by the authors, 2017

While FunG has several unique features it also lacks some functionality provided by other AD-implementations. In particular it currently does not provide reverse-mode automatic differentiation. Due to its generic approach it also lacks full jacobian computation, which would require the introduction of new customized data structures, i.e. for higher order tensors. This strongly opposes one of the principal ideas behind FunG, namely providing an abstraction layer that still allows each user to use his own, possibly optimized, data structures[4].

To keep the presentation concise we will mainly restrict the discussion to scalar variables. Extensions for matrices are discussed where necessary. The implementation strictly follows the underlying mathematical structure. This is explained in Sec. 2. Then provided mathematical functions, such as common trigonometric functions and matrix invariants, are shortly introduced in Sec. 3. In Sec. 4 we will discuss different optimization strategies that are implemented in FunG, using two invariant-based models as examples. Then, in Sec. 5, the performance of FunG is compared with several automatic differentiation (AD) libraries. In Sec. 6 examples that illustrate several features of FunG are given. This paper ends with a short summary of the attained results (Sec. 7) and instructions for downloading and installing the library (Sec. 8).

2 Concepts

We begin by introducing the main concepts and techniques that are necessary for an efficient and generic implementation. For this first the interfaces for functions and variables are introduced. Then we discuss the heart of FunG, the implementations of elementary differentiation rules.

2.1 Functions in FunG

In FunG every nullary callable is treated as a constant function. Non-constant functions additionally provide a function `update` to change the point of evaluation $x \in X$, where X denotes some vector space. Optionally up to the first three directional derivatives may be specified. The interface for functions is summarized in the following list:

- Evaluate $f(x)$, where x has been assigned before:

C++ code
```
1 template <class Arg>
2 const auto& operator()() const;
```

- Set the point of evaluation x for general arguments (for non-constant functions):

C++ code
```
1 template <class Arg>
2 void update(const Arg& x);
```

- Evaluate the first directional derivative $f'(x)\delta x$ (for non-constant functions):

C++ code
```
1 template <class Arg>
2 auto d1(const Arg& dx) const;
```

- Evaluate the second directional derivative $f''(x)(\delta x, \delta y)$ (for non-linear functions):

C++ code
```
1 template <class ArgX, class ArgY>
2 auto d2(const ArgX& dx, const ArgY& dy) const;
```

[4]While not providing this functionality in a generic setting it is straightforward to write the corresponding assembly-loop using FunG.

© by the authors, 2017

- Evaluate the third directional derivative $f'''(x)(\delta x, \delta y, \delta z)$ (for non-linear, non-quadratic functions):

C++ code

```
1    template <class ArgX, class ArgY, class ArgZ>
2    auto d3(const ArgX& dx, const ArgY& dy, const ArgZ& dz) const;
```

Derivatives that are not implemented are interpreted as functions that always return zero. This design decision has significant advantages during code generation, see Sec. 4.2. However, it is not very convenient to use a function that may or may not provide certain member functions. Thus it is advisable to always use a *finalized version* of the generated function, i.e. instead of the generated function `f`, better use the result of `finalize(f)`. The call to `finalize` adds all undefined derivatives and, depending on the generated function, may perform some simplifications of the interface. This is explained in see Sec. 6. [5]

2.2 Function arguments in FunG

In the last section an interface for functions on vector spaces was presented. Therefore function arguments should at least satisfy the requirements of a vector space element, they should be scalable and sumable. For non-scalar variables additional operations should be supported, such as matrix-matrix- or matrix-vector-multiplication. As example consider the matrix-valued example in the introduction, where scalability, sumability and matrix-matrix-multiplication is required for the function arguments.

In the following subsections the required interfaces for the classical function arguments scalars, vectors and matrices are summarized. Other vector space elements are possible, but must be provided by the user together with corresponding functions.

2.2.1 General interface. The following requirements hold for all variables that are to be used with FunG:

- one of the following operations must be valid for variables `v,w` of type `V`:

C++ code

```
1 v += w;
2 V x = v + w;
```

If only the first alternative is available, then FunG will generate an implementation of the free summation operator.

- one of the following operations must be valid for `a` of arithmetic type `A` [6]:

C++ code

```
1 v *= a;
2 V w = a*v;
```

If only the first alternative is available, then FunG will generate an implementation of the free multiplication operator.

[5] Currently no derivatives or order higher than the third are implemented. This is due to two reasons. First, for solving optimization problems typically no more than the first two or three derivatives are required. Secondly, derivatives of arbitrary order require to additionally pass the order as template argument. Together with the increasing number of template and function arguments this leads to signatures that are sufficiently hard to read to justify the introduction of a proxy object to increase readability. This in turn leads to a less intuitive syntax.

[6] Here “arithmetic” is used in the sense that `std::is_arithmetic<A>::value == true`.

 © by the authors, 2017

These operations are not only available for built-in arithmetic types, but at least one of the alternatives is implemented in most C++ matrix libraries, such as Eigen (Guennebaud and Jacob [2010]), Dune-ISTL (Blatt and Bastian [2007]) or Armadillo (Sanderson [2010]).

2.2.2 Extended interface.

For matrices and vectors additional functionality is required. Since vectors are treated analogously, the discussion is restricted to matrices. For these the additional requirements are:

- access to entries,
- access to the number of rows and
- access to the number of columns.

For all cases there exist free template functions that provide access to the desired quantity via traits classes. For accessing matrix entries there is:

C++ code

```
template <class Matrix, class Index,
          class = std::enable_if_t<std::is_integral<Index>::value> >
decltype(auto) at( Matrix&& A, Index i, Index j )
{
  return AccessEntryOfMatrix<Matrix>::apply( A, i, j );
}
```

Currently, FunG supports the most popular ways of accessing matrix entries, via:

`A[i][j]` or `A(i,j)`.

Using the SFINAE[7]-technique of Sec. 2.4 it is easy to choose the correct implementation at compile time [8].

Regarding the access of the number of rows, resp. columns, matrices of constant size and of dynamic size are distinguished. For constant size matrices access to the number of rows and columns at compile time enables loop-unfolding optimizations for the C++-compiler. To exploit this fact, slightly different interfaces are provided for matrices of constant and dynamic size.

2.2.2.1 Matrices of constant size.

For constant size matrices the following free template functions admit access to the number of rows resp. columns:

C++ code

```
template <class Matrix>
constexpr auto rows()
{
  return NumberOfRows<Matrix>::value;
}

template <class Matrix>
constexpr auto cols()
{
  return NumberOfColumns<Matrix>::value;
}
```

Matrices of constant size often take the numbers of rows and columns as template parameters. If template matrix types can be specified with one of the patterns

[7] Substitution Failure Is Not An Error.

[8] In case that both ways of accessing entries are supported, square brackets are used.

C++ code

```
using M1 = MyMatrix<nRows,nCols>
// or
using M2 = MyMatrix<Scalar,nRows,nCols>
```

then FunG will automatically detect the number of rows and columns. Else you need to provide a specialization of the template classes `NumberOfRows` and `NumberOfColumns`. To illustrate usage we consider the implementation of the scalar product $(A, B) := \sum_{i,j} A_{ij}B_{ij}$:

C++ code

```
template <class Matrix>
auto scalarProduct(const Matrix& A, const Matrix& B)
{
  using Index = decltype( rows<Matrix>() );

  auto result = decltype( at(A,0,0) )(0);
  for( Index i = 0 ; i < rows<Matrix>() ; ++i )
    for( Index j = 0 ; j < cols<Matrix>() ; ++j )
      result += at(A,i,j) * at(B,i,j);

  return result;
}
```

2.2.2.2 Matrices of dynamic size. For matrices of dynamic size the following free template functions admit access to the number of rows resp. columns:

C++ code

```
template <class Matrix>
auto rows(const Matrix& A)
{
  return DynamicNumberOfRows<Matrix>::apply( A );
}

template <class Matrix>
auto cols(const Matrix& A)
{
  return DynamicNumberOfColumns<Matrix>::apply( A );
}
```

Currently, there is support for

- access via public member variables `n_rows` and `n_cols` (as used in Armadillo),
- access via public member functions `rows()` and `cols()`.

For other signatures suitable specializations of the template classes `DynamicNumberOfRows` and `DynamicNumberOfColumns` must be provided. Again, we illustrate usage with the implementation of the scalar product $(A, B) := \sum_{i,j} A_{ij}B_{ij}$:

C++ code

```
template <class Matrix>
auto scalarProduct(const Matrix& A, const Matrix& B)
{
  assert( rows(A) == rows(B) && cols(A) == cols(B) );
  using Index = decltype( rows(A) );

  auto result = decltype( at(A,0,0) )(0);
  for( Index i = 0 ; i < rows(A) ; ++i )
    for( Index j = 0 ; j < cols(B) ; ++j )
      result += at(A,i,j) * at(B,i,j);

```

 © by the authors, 2017

```
12    return result;
13  }
```

2.2.3 Usage of non-built-in arithmetic types. FunG admits the usage of user-defined arithmetic types. For this a suitable specialization of the template class `IsArithmetic` must be provided:

C++ code

```
1  template <>
2  struct IsArithmetic< UserDefinedScalarType >
3    : std::true_type
4  {};
```

2.2.4 Working with expression templates in matrix libraries. In some matrix-libraries, such as Eigen (cf. Guennebaud and Jacob [2010]), mathematical operations are implemented with expression templates that delay the actual computation until the next assignment. This may interfere with generic implementations that use automatic type deduction. To avoid these problems in FunG, a suitable specialization of the template class `Decay` must be provided. This is illustrated for the Eigen-library, where expression-templates provide the underlying matrix- or vector-type with the nested type `PlainObject`.

C++ code

```
1  // Default case (identity).
2  template < class F, class = void >
3  struct Decay
4  {
5    using type = F;
6  };
7
8  // Underlying type for expression templates of the Eigen library.
9  template < class F >
10 struct Decay< F, void_t< Checks::TryNestedType_PlainObject<F> > >
11 {
12   using type = typename F::PlainObject;
13 };
```

2.2.5 Multiple variables To work with multiple variables FunG provides a special template class to associate an id with a variable:

C++ code

```
1  template <class Type, int id>
2  class Variable;
```

This class is a function in the sense of 2.1. If derivatives are computed with respect to `id`, then this class behaves like a linear function, else like a constant function. This association of ids with variables at compile time has two advantages. First, it admits the generation of highly efficient code. This is explained in sec 4.2. Secondly, this approach does not require any additional storage for `id`.

We can use automatic type deduction to simplify the creation of a new variable:

C++ code

```
1  // create variable of type Variable<std::decay_t<decltype(x)>,0>
2  auto v0 = variable<0>(x)
```

For the template argument `Type` any type that satisfies the interface for variables is admissible, see 2.2. For problems that only depend on one variable the template class `Variable` is not required.

2.3 Mathematical operations

Having defined requirements on functions and their arguments we now turn to the question how to build up complex functions from elementary ones. Exemplarily, we consider

$$h(x) = \exp(\sin(x)) + \exp(x)\log(x) = (f_0 \circ f_1)(x) + f_0(x)f_2(x), \tag{1}$$

with $f_0(x) = \exp(x)$, $f_1(x) = \sin(x)$ and $f_2(x) = \log(x)$. The representation of h as combination of more elementary functions can be represented in a tree structure. This is illustrated in Fig. 1:

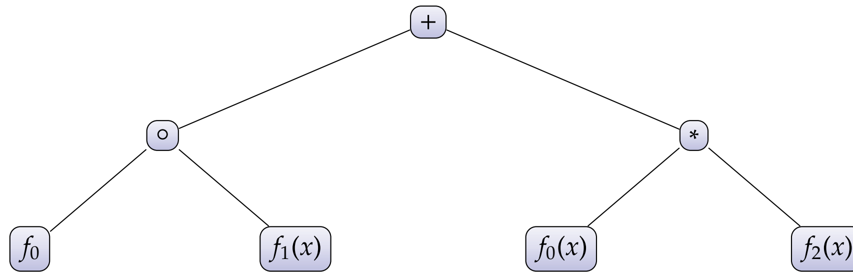


Figure 1: Tree structure associated with $h(x) = \exp(\sin(x)) + \exp(x)\log(x) = (f_0 \circ f_1)(x) + f_0(x)f_2(x)$.

The leafs are functions and all other nodes represent a mathematical operation. Then, the differentiation process can be described by a small set of operations that generate a new tree. These operations are equivalents of mathematical differentiation rules:

- the *sum rule*,

$$(f(x) + g(x))' = f'(x) + g'(x)$$

replaces the child nodes with its derivatives:

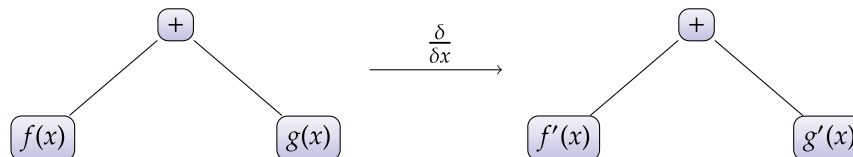


Figure 2: Tree transform associated with the sum rule.

- the *product rule*,

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$$

generates a more complex tree:

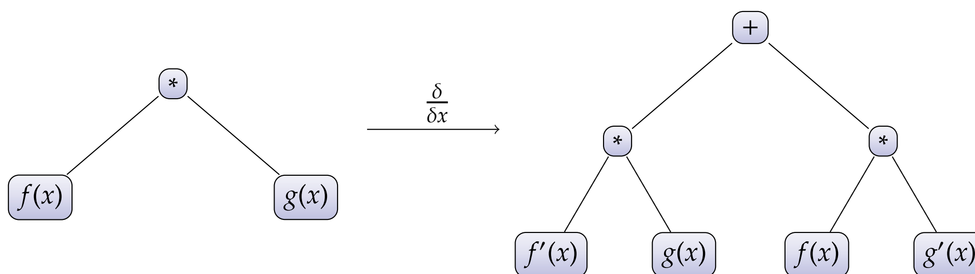


Figure 3: Tree transform associated with the product rule.

 © by the authors, 2017

- and the *chain rule*,

$$f(g(x))' = f'(g(x))g'(x)$$

yields:

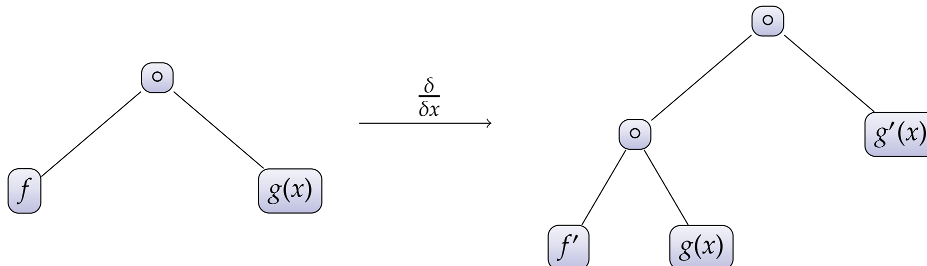


Figure 4: Tree transform associated with the chain rule.

With the given transformation rules it is easy to get the derivative of an arbitrary complex function by traversing its tree representation from top to bottom and repeatedly applying the differentiation rules. For the derivative of the function h, defined at the start of this subsection, we get:

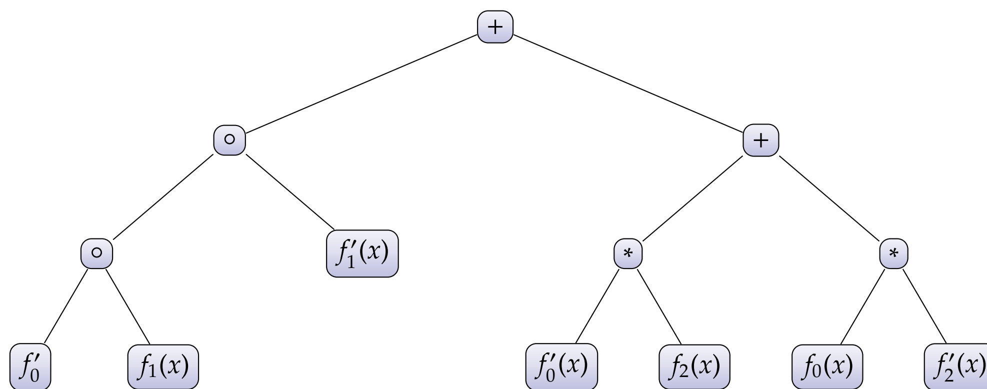


Figure 5: Example tree structure corresponding to $h'(x)$.

2.3.0.1 Expression templates. We can easily evaluate the derivative $h'(x)$ by traversing the generated tree. However, setting up such a tree structure and traversing it adds significant overhead that will slow down computations significantly. The classical technique for avoiding this overhead is based on expression-templates, cf. Veldhuizen [1995], and used in most AD-implementations, cf. eg. Hogan [2014], Walther and Griewank [2012], Sagebaum et al., Bendtsen and Stauning [1996], Heroux et al. [2005].

Expression-templates provide an abstraction layer that separates the interface for, i.e. mathematical, operations from its implementation. With this abstraction the evaluation of the tree structures can be performed at compile time.

In the context of FunG, simplified versions of the mathematical operators take the forms:

C++ code

```
1 template <class F, class G,
2           /* static concept checking */>
3 auto operator+(const F& f, const G& g) {
4   return MathematicalOperations::Sum<F,G>(f,g);
5 }
```

C++ code

```
template <class F, class G,
          /* static concept checking */>
auto operator*(const F& f, const G& g) {
  return MathematicalOperations::Product<F,G>(f,g);
}
```

The template classes Sum and Product hide the actual implementation of the mathematical operations and provide the corresponding differentiation rules. Additionally there exist implementations of operator- based on operator+.

For the chain rule an overload of the function call operator is required. This overload must be added to each function that is used within FunG (at least if the chain rule is used). This can be done by using the CRTP[9]-based class Chainer:

C++ code

```
template <class Function>
struct Chainer
{
  // provide function call operator
  decltype(auto) operator()() const noexcept
  {
    return static_cast<const Function*>(this)->d0();
  }

  // support chain rule
  template <class OtherFunction,
            /* static concept checks */>
  auto operator()(const OtherFunction& g)
  {
    return MathematicalOperations::Chain<Function,OtherFunction>
        (*static_cast<const Function*>(this),g);
  }

  ...
};
```

To avoid explicitly importing the overloads of Chainer to the derived class this class should no more define the function call operator. Instead it should define a function d0 with same signature as the function call operator would have.

Assume that we already have defined a class Cos satisfying the function interface but not yet the chain rule. Then the easiest way to add support for the chain rule is to change it from

C++ code

```
class Cos {
  ...
  double operator() const noexcept;
  ...
};
```

to

C++ code

```
#include ``fung/util/chainer.hh''

class Cos : public Chainer<Cos> {
  ...
  double d0() const noexcept;
  ...
};
```

[9] Curiously Recurring Template Pattern.

 © by the authors, 2017

As example we consider the function h given at the beginning of this section. In FunG we can write

C++ code

```
auto x = variable<0>(1.); // point of evaluation
auto fun = finalize( exp(sin(x)) + exp(x)*log(x) );
```

to generate an object of type

C++ code

```
Sum<
    Chain<Exp,Sin>,
    Product<Exp,Log>
    >
```

This class contains all relevant information to generate the derivatives of `fun`. Thus the tree structures described in the beginning of this section are generated and resolved at compile time. Since all type-information is provided to the compiler most function calls will be eliminated when resolving these trees. This approach allows to avoid almost all implementation-related runtime overhead.

2.4 SFINAE and void_t

In FunG SFINAE-techniques based on `void_t`, cf. Brown [2014], are heavily used. `void_t` is a template alias that is equivalent to `void` for all *valid* template arguments[10]:

C++ code

```
template <class...>
using void_t = void;
```

This alias matches any well-formed type into the predicable type `void`. In the context of FunG, this allows to verify interfaces at compile-time, provide a flexible interface to matrix and vector implementations as well as to remove redundant code paths from the evaluation of derivatives.

This is illustrated for the case of the first derivative in FunG, i.e. the case that we want to know if objects of type `T` provide a member function `d1`, taking one argument of type `Arg`. We can try to access this member function in an unevaluated context via

C++ code

```
template <class T, class Arg>
using TryMemFn_d1 =
  decltype( std::declval<T>().d1(std::declval<Arg>()) );
```

This alias can be used to define a meta-function as follows:

C++ code

```
// Default case, no suitable member function d1.
template <class T, class Arg, class=void>
struct HasMemFn_d1
  : std::false_type
{};

// Specialization for the case that a suitable member function d1 exists.
template <class T, class Arg>
struct HasMemFn_d1< T , Arg , void_t< TryMemFn_d1<T,Arg> > >
  : std::true_type
{};
```

[10] This implementation of `void_t` does not directly work with gcc-x, x<5. For these compiler versions another level of indirection must be added.

© by the authors, 2017

This meta-function can be used to choose different code paths at compile time. Alternatively one can use the same technique as in the specialization of `HasMemFn_d1` to directly use SFINAE for the choice of implementations.

2.5 Error handling

Debugging errors in strongly templated C++ code is often not easy due to the hard-to-read compiler output. Two ways of providing meaningful error messages are provided. Using the technique described in the last section, we can check satisfaction of the required interfaces and consistency of the defined function at compile time. Using `static_assert` meaningful compiler error messages are provided.

Sometimes things may go wrong because of the concatenation $f \circ g$ of two functions, where $\operatorname{ran}(g) \not\subseteq \operatorname{dom}(f)$. If f is implemented with the help of functions from `cmath`, then one can inspect the corresponding error flag. Alternatively one may define the macro `FUNG_ENABLE_EXCEPTIONS`. If defined, it enables an `OutOfDomainException` that is thrown as soon as a function argument leaves the admissible domain.

3 Provided functions

Mny functions that are contained in `cmath`, as well as matrix invariants and an implementation of the strain tensor are provided in FunG.

3.1 Functions from `cmath`

The functions in `cmath` are contained in namespace `std`. Though, in the `C++11`-standard it is allowed that the implementation just forwards to the `C` math library in `math.h`, wherein the functions and macros are part of the global namespace. This practice is followed in most `stl`-implementations. For this reason it is not possible to provide the same signature for built-in arithmetic types.

A common strategy to solve this problem in AD-implementations is the introduction of a special variable, such as `adouble` in ADOL-C. In FunG the template class `Variable` can be used for this purpose. For problems that only depend on one variable, it is also admissible to directly use custom types[11]. To avoid ambiguity for functions that are contained in `cmath` the first letter of the function name must be changed to uppercase:

C++ code

```
1 auto f = Sin(1.)                // uppercase to avoid ambiguity
2 auto g = sin( variable<0>(1.) ) // no ambiguity
```

An overview on the currently available scalar functions is given in Table 1.

3.2 Invariants

Since physical models should be independent of the position of the observer they are formulated in terms of invariants, such as $\det(A)$, $\operatorname{tr}(A)$ or $\operatorname{cof}(A)$. In FunG, optimized implementations of the determinant[12], the trace and the cofactors are provided for both, matrices of constant and dynamic size. Based on these, implementations of different invariants are provided. These are summarized in table 2.

The first and third invariant, the trace and determinant of a matrix $A \in \mathbb{R}^{n,n}$, can also be accessed via `trace(A)` resp. `det(A)`. Another important quantity in invariant-based modeling is the strain

[11] In this case the post-processing through `finalize` yields a simpler interface, cf. Sec. 6.

[12] The determinant is implemented for matrices $A \in \mathbb{R}^{n,n}$, $n = 2, 3$.

 © by the authors, 2017

Table 1: Available scalar functions in FunG.

Formula	Code	Code (built-in arithmetic)
$x^{n/m}$	`pow<n,m>(x)`	`Pow<n,m>(x)`
$\sqrt{x}$	`sqrt(x)`	`Sqrt(x)`
$\sqrt[3]{x}$	`cbrt(x)`	`Cbrt(x)`
$\sqrt[3]{x^2}$	`cbrt2(x)`	`Cbrt2(x)`
2^x	`exp2(x)`	`Exp2(x)`
$\exp(x)$	`exp(x)`	`Exp(x)`
$\log(x)$	`log10(x)`	`Log10(x)`
$\log_2(x)$	`log2(x)`	`Log2(x)`
$\ln(x)$	`ln(x)`	`LN(x)`
$\sin(x)$	`sin(x)`	`Sin(x)`
$\cos(x)$	`cos(x)`	`Cos(x)`
$\tan(x)$	`tan(x)`	`Tan(x)`
$\arcsin(x)$	`asin(x)`	`ASin(x)`
$\arccos(x)$	`acos(x)`	`ACos(x)`

Table 2: Available invariants in FunG.

Invariant	Formula	Code
$\iota_1(A)$	$= \mathrm{tr}(A)$	`i1(A)`
$\iota_2(A)$	$= \mathrm{tr}(\mathrm{cof}(A))$	`i2(A)`
$\iota_3(A)$	$= \det(A)$	`i3(A)`
$\iota_4(A,M)$	$= \mathrm{tr}(AM)$	`i4(A,M)`
$\iota_5(A,M)$	$= \mathrm{tr}(AM^2)$	`i5(A,M)`
$\iota_6(A,M)$	$= \mathrm{tr}(A^2M)$	`i6(A,M)`
$\bar{\iota}_1(A)$	$= \mathrm{tr}(A)\det(A)^{-1/3}$	`mi1(A)`
$\bar{\iota}_2(A)$	$= \mathrm{tr}(\mathrm{cof}(A))\det(A)^{-2/3}$	`mi2(A)`
$\bar{\iota}_4(A,M)$	$= \mathrm{tr}(M,A)\det(A)^{-1/3}$	`mi4(A,M)`
$\bar{\iota}_5(A,M)$	$= \mathrm{tr}(AM^2)\det(A)^{-1/3}$	`mi5(A,M)`
$\bar{\iota}_6(A,M)$	$= \mathrm{tr}(A^2M)\det(A)^{-2/3}$	`mi6(A,M)`

tensor $C = F^TF$ for $F \in \mathbb{R}^{n,n}$. An efficient implementation, that exploits symmetry, can be accessed via:

C++ code

```
1 auto C = strainTensor(F);
```

4 Performance optimization

FunG was developed with focus on integrands that are evaluated in finite-element assembly procedures. Thus performance is important. In this section the three most relevant optimization strategies are shortly presented. Two examples from biomechanics are used to illustrate their effects. The first is a compressible version of the muscle model which has been shown in the

© by the authors, 2017

introduction:

$$
\begin{aligned}
W(F) = {} & c\left[\exp(b(\bar{\iota}_1(C)-3))-1\right] + d\left[\exp(a(\bar{\iota}_6(C,M)-1)^2)-1\right] + \\
& e\det(C) + \frac{f}{2}\log(\det(C)), \\
\bar{\iota}_1 &= \operatorname{tr}(C)\det(C)^{-1/3}, \qquad C = F^T F, \\
\bar{\iota}_6 &= \operatorname{tr}(CM^2)\det(C)^{-1/3},
\end{aligned}
$$

where a, b, c, d, e and f are material parameters. In FunG this can be implemented as follows:

C++ code

```
// for given MatrixType
// initialize the structural tensor M
auto M = LinearAlgebra::unitMatrix<Matrix>();
// initialize the deformation gradient F
auto F = LinearAlgebra::unitMatrix<Matrix>();

auto model = c*( exp( b*( mi1(F) - 3 ) ) - 1 )
           + d*( exp( a*pow<2>( mi6(F,M) - 1 ) ) - 1 )
           + e*det(F) + f/2*log(det(F));

// generate strain tensor
auto C = strainTensor(F);
// generate material model W(C)
auto W = model( C );
```

In the above model `mi1` and `mi6` denote the first modified invariant $\bar{\iota}_1(F)$, resp. the third modified mixed invariant $\bar{\iota}_6(F, M)$ and the function `strainTensor(F)` generates the strain tensor $C = F^T F$.

The second is a compressible model for adipose tissue from Sommer et al. [2013]:

$$
\begin{aligned}
W(F) = {} & c_{\text{Cells}}(\iota_1 - 3) + \frac{k_1}{k_2}\exp(k_2(\kappa\iota_1 + (1-3\kappa)*\iota_4)^2 - 1) + \\
& e\det(C) + \frac{f}{2}\log(\det(C)), \\
\iota_1 &= \operatorname{tr}(C), \qquad C = F^T F, \\
\iota_4 &= \operatorname{tr}(CM),
\end{aligned}
$$

where c, k_1, k_2, κ are material parameters. Its implementation in FunG is given through

C++ code

```
// for given MatrixType
// initialize the structural tensor M
auto M = LinearAlgebra::unitMatrix<Matrix>();
// initialize the deformation gradient F
auto F = LinearAlgebra::unitMatrix<Matrix>();

auto model = c*( i1(F) - 3 )
           + d*( exp( a*pow<2>( mi6(F,M) - 1 ) ) - 1 )
           + e*det(F) + f/2*log(det(F));

// generate strain tensor
auto C = strainTensor(F);
// generate material model W(C)
auto W = model( C );
```

In the following all computations are performed on an ASUS UX32V, 4xi7-3517 1.90 GHz, 10 GiB DDR3-RAM 1600 MHz under Kubuntu 16.04. The code was compiled with gcc 5.3.1 and additional compiler flag `-std=c++1y`. Constant-size matrices of Eigen, version 3.2.0, see Guennebaud

 © by the authors, 2017

and Jacob [2010] are used. For each described optimization technique the average computation time for the evaluation of single member functions will be given, where the average is computed over 10^7 evaluations at varying points of evaluation. No numbers are given for the evaluation of the function value. This is due to the fact that the related computations are fully performed in the `update`-function, see Sec. 4.1. Thus the function call operator only has to access the cached value. Recall that the performance of FunG is, to large parts, a consequence of compiler-friendly code design. Thus all measurements are strongly dependent on the used compiler.

4.1 Caching

The most efficient strategy to reduce the computation time in automatic differentiation is caching. This is mainly due to two reasons. First, caching intermediate results that are expensive to compute pays off. This does not only include caching of the results of iterative computations such as `sqrt(x)`, `sin(x)` or `exp(x)`, but also the direct evaluation of every function in its `update`-function. This avoids repeated evaluation in expressions such as

$$(f(x) * g(x) * h(x))' = f'(x)g(x)h(x) + f(x)g'(x)h(x) + f(x)g(x)h'(x),$$

where each function value is used twice. Secondly this admits to reduce the number, and often also the size, of temporary variables. Both effects help avoiding expensive cache misses. The influence on performance is illustrated in table 3.

	adipose tissue				muscle tissue			
caching	update	d1	d2	d3	update	d1	d2	d3
with	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
without	270 ns	190 ns	714 ns	3090 ns	473 ns	353 ns	1673 ns	11907 ns
ratio	0.96	0.23	0.22	0.16	01.00	0.18	0.18	0.15

Table 3: Average computation time with and without caching.

4.2 Elimination of zeros

One of the disadvantages of automatic differentiation is the fact that for higher-order, possibly mixed, derivatives significant computation time can be spend with the computation of zeros. In most cases the compiler is not able to remove these redundant operations. Thus it pays off to implement a mechanism that eliminate these branches.

The essential idea is to not implement functions that always return zero (such as the second derivative of a linear function). Using the technique described in Sec. 2.4 it is straightforward to provide suitable optimizations for the mathematical operations described in Sec. 2.3 [13].

The impact on performance is illustrated in table 4. For the `update`- and the `d1`-function the observed differences are negligible. This is to be expected since no operation can be eliminated from the first and only few from the latter. For higher derivatives the elimination of zeros becomes more and more relevant.

[13] This design decision also enables efficient treatment of multiple variables and mixed derivatives.

	adipose tissue				muscle tissue			
zeros	update	d1	d2	d3	update	d1	d2	d3
with	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
without	267 ns	42 ns	192 ns	835 ns	477 ns	60 ns	362 ns	2333 ns
ratio	0.97	1.05	0.83	0.59	0.99	1.03	0.84	0.75

Table 4: Average computation times with and without elimination of zeros.

4.3 Compiler flags

FunG largely consists of template classes, functions and aliases. Thus one of the prerequisites for the generation of efficient code is that the compiler must inline most of the function calls. In particular significantly larger amounts of code than in the "average" `C++`-code must be inlined. In invariant-based models, where also loops over matrix entries must be unfolded, significant performance increases can often be observed after adjusting the following compiler parameters:

- `early-inlining-insns=5000`
- `max-inline-insns-auto=3000`
- `inline-unit-growth=100`

For more details, see Stallman and the GCC Developer Community [2015], Alexandrescu [2014]. The influence of these adjustments is illustrated in table 5.

	adipose tissue				muscle tissue			
flags	update	d1	d2	d3	update	d1	d2	d3
with	210 ns	3 ns	75 ns	166 ns	400 ns	3 ns	246 ns	1369 ns
without	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
ratio	0.81	0.07	0.47	0.34	0.85	0.05	0.81	0.78

Table 5: Average computation times with and without additional compiler flags.

Note that there is no golden rule for choosing compiler parameters. Its effects strongly differ between compilers and compiler versions. Moreover, when compiled together with an application, where FunG typically only is a small part, other parts of the code may influence the way the compiler optimizes the code. The latter effect can be avoided by compiling function definitions separately.

5 Comparison with AD-libraries

To give a first idea of the performance of FunG it is compared with several AD-libraries. For this, we consider the forward mode AD-schemes of

- ADOLC, version 2.6.1, see Walther and Griewank [2012],
- Adept, version 1.1, see Hogan [2014],
- CoDiPack, version 1.2.1, see Sagebaum et al.,

 © by the authors, 2017

- CppAD, github, master branch from 01/23/2016, see Bell,
- FADBAD++, version 2.1, see Bendtsen and Stauning [1996] and
- Sacado, version 12.4.2 of Trilinos, see Heroux et al. [2005].

Again the computations are performed on an ASUS UX32V, 4xi7-3517 1.90 GHz, 10 GiB DDR3-RAM 1600 MHz under Kubuntu 16.04. The code was compiled with gcc 5.3.1 and additional compiler flag `-std=c++1y`.

Since not all investigated libraries support higher derivatives only the first derivative is considered. Moreover, the following examples are restricted to scalar variables. The reason is that matrix-valued variables are not directly supported by the other libraries [14].

In tables 6-8 computation times for different functions are compared with the best manual implementation that the author was able to provide for both, the function value and the first derivative. Ratios are given with respect to this manual implementation. Since not all libraries support the separate evaluation of function value and derivative the average evaluation time of both quantities is measured over 10^7 evaluations with different function arguments.

- The first example is

$$f_0(x) = x^{3/2} + \sin(\sqrt{x}).$$

This function can also be written as $f_0 = g \circ h$, where $g(x) = x^3 + \sin(x)$ and $h(x) = \sqrt{x}$. The latter implementation only requires one evaluation of $\sqrt{x}$. In table 6 the corresponding average computation time is given under *FunG (opt)*. From the same table we see that a straight-forward implementation of f_0 in FunG is about 10% slower, since $\sqrt{x}$ must be evaluated twice.

Manual	FunG (opt)	FunG	CoDiPack	Sacado	FADBAD	Adept	ADOLC	CppAD
60 ns	60 ns	65 ns	65 ns	68 ns	222 ns	304 ns	401 ns	926 ns
1.00	1.00	1.08	1.08	1.13	3.70	5.07	6.68	15.4

Table 6: Comparison of the average computation time for the evaluation of function value and derivative for $f_0(x) = x^{3/2} + \sin(\sqrt{x})$. Ratios are given with respect to the manual implementation.

As illustrated in table 6, FunG achieves the same performance as the optimized manual implementation. Without the above introduced reformulation of f_0 FunG provides the same performance as CoDiPack. Sacado is slighly slower in this example. Despite the fact that f_0 is a very simple function the remaining libraries, FADBAD++, Adept, ADOLC and CppAD are already significantly slower. These differences will be even more pronounced in the following examples.

- The next example is

$$f_1(x) = 1 + x(1 + x(1 + x(1 + x)))$$
$$(= \sum_{i=0}^{4} x^i)$$

This example only consists of simple and cheap operations. In particular nothing has to be computed iteratively. As illustrated in table 7, both FunG and CoDiPack achieve the same performance as the optimized manual implementation. All other implementations are significantly slower.

[14] In case that vector-valued variables are supported one could work with vectorized matrices.

Manual	FunG	CoDiPack	Sacado	FADBAD	Adept	ADOLC	CppAD
1.7 ns	1.7 ns	1.7 ns	4.0 ns	13.9 ns	45.8 ns	336 ns	558 ns
1.00	1.00	1.00	2.35	8.18	26.9	198	328

Table 7: Comparison of the average computation time for the evaluation of function value and derivative for $f_1(x) = 1 + x(1 + x(1 + x(1 + x)))$. Ratios are given with respect to the manual implementation.

- The last example is given through

$$f_2(x, y, z) = xyz.$$

This demonstrates the effect of multiple variables[15].

Manual	FunG	CoDiPack	FADBAD	Sacado	Adept	ADOLC	CppAD
1.7 ns	1.7 ns	2.0 ns	8.1 ns	40 ns	48 ns	877 ns	1781 ns
1.00	1.00	1.18	4.76	23.5	28.2	515	1047

Table 8: Comparison of the average computation time for the evaluation of function value and derivative for $f_2(x, y, z) = xyz$. Ratios are given with respect to the manual implementation.

As illustrated in table 8 FunG achieves the same performance as the optimized manual implementation. Only the forward-scheme of CoDiPack, which is 18% slower, provides roughly comparable performance. The large differences with respect to the other examined libraries is partly a consequence of the elimination strategy of Sec. 4.2.

6 Examples

We provide some examples that illustrate different features and how to use FunG. In Sec. 6.1 a model for adipose tissue is described. It illustrates the generation of invariant-based models without usage of the template class `Variable`. In Sec. 6.2 a model with different variables of different types is generated. Eventually, a model of nonlinear heat transfer is described. Such models depend on the state and the gradient of the heat distribution. How this is realized in FunG is demonstrated Sec. 6.3.

6.1 A model for adipose tissue

We begin with the model for adipose tissue of Sommer et al. [2013], which was already used in Sec. 4. It is assumed that the number of rows/columns of the used matrices can be deduced at compile time. Here $c_{Cells}, k_1, k_2, \kappa$ denote material parameters, F is the deformation gradient and M is a structural tensor describing the fiber directions of the interlobular septa [16]. The functions `i1` and `i4` implement the first invariant $\iota_1(F) = \mathrm{tr}(F)$, resp. the first mixed invariant $\iota_4(F, M) = \mathrm{tr}(FM)$.

[15] The compared libraries provide different ways of computing derivatives in vector mode. Adept computes the adjoint with respect to the Euclidean scalar product, CppAD computes the gradient and all others directly compute the directional derivatives. Since all three approaches require the computation of the same values, the author considers these slightly differing approaches worthy to compare.

[16] In adipose (fat) tissue anisotropic properties are mainly attributed to the interlobular septa, a network of long fibrous collagen bundles, see. Sommer et al. [2013].

 © by the authors, 2017

C++ code

```
template <class Matrix>
auto adiposeTissueModel(double cCells, double k1, double k2,
                        double kappa, const Matrix& M, const Matrix& F,
                        int n = dim<Matrix>()) {
  using namespace FunG;
  using namespace FunG::LinearAlgebra;

  auto aniso = kappa*i1(F) + ( 1 - 3*kappa ) * i4(F,M) - 1;
  auto materialLaw = cCells*( i1(F) - n ) + (k1/k2) * ( exp(k2*pow<2>(aniso)) - 1 );

  return finalize( materialLaw( strainTensor(F) ) );
}

auto f = adiposeTissueModel(cCells,k1,k2,kappa,M,F);
```

Observe that no specific variable is associated with the unknown F. In this case the call to `finalize` yields a simplified interface. For some deformation F, the point of evaluation can be changed via:

C++ code

```
 f.update(F);
```

Access to function value and derivatives, with perturbations $dF0, dF1, dF2$, is given through:

C++ code

```
auto value            = f();
auto firstDerivative  = f.d1(dF0);
auto secondDerivative = f.d2(dF0,dF1);
auto thirdDerivative  = f.d3(dF0,dF1,dF2);
```

Avoidance of the template class `Variable` enables another function that may be convenient. Namely a call to `finalize` adds the additional member function:

C++ code

```
template < class Arg >
auto operator()( Arg&& x ) {
  update( std::forward<Arg>( x ) );
  return operator()();
}
```

Thus it is also possible to write

C++ code

```
auto value = f(F);
```

instead of

C++ code

```
f.update(F);
auto value = f();
```

6.2 An example with two variables

In this example we consider a function that takes a scalar and a matrix-valued argument:

$$f(x, M) = \sqrt{x} * \mathrm{tr}(M).$$

To generate f we use the following code:

© by the authors, 2017

C++ code

```
#include ''fung.hh''

template <class Matrix>
auto myFunction(double initialX, const Matrix& initialM) {
  using namespace FunG;
  using FunG::LinearAlgebra::trace;

  auto x = variable<0>(initialX);
  auto M = variable<1>(initialM);

  return finalize( sqrt(x)*trace(M) );
}
...
f = myFunction(1.,M0);
```

Then, for scalar variables x, $dx1$, $dx2$ and matrix-valued variables M, dM we can change the point of evaluation via:

C++ code

```
f.template update<0>(x);
f.template update<1>(M);
```

Access to the function value and its derivatives is given through:

C++ code

```
double value        = f();
double df_dx        = f.template d1<0>(dx1);
double df_dM        = f.template d1<1>(dM);
double ddf_dMdx     = f.template d2<1,0>(fM,dx1);
double dddf_dxdxdM = f.template d3<0,0,1>(dx1,dx2,dM);
```

6.3 A model for nonlinear heat transfer

Nonlinear partial differential equations require another feature, the possibility to use different types associated to the same variable, such as representations of the value and gradient of a variable. A popular example for these problems are models of nonlinear heat transfer such as the following:

$$A(u, \nabla u) = (1 + |u|^2)\nabla u$$

When computing $A'(u, \nabla u)\delta u$, then both u and ∇u must be taken into account. For this we need to assign the same variable id to both value and gradient:

C++ code

```
#include ''fung.hh''

template <class Scalar, class Vector>
auto heatModel(const Scalar& u0, const Vector& du0) {
  using namespace Fung;

  auto u = variable<0>(u0);   // state variable
  auto du = variable<0>(du0); // gradient variable, both with id 0

  return finalize( ( 1 + pow<2>(u) )*du );
}

auto f = heatModel(u0,du0);
```

To work with this kind of model arguments must be packed into tuples. Then, for a variable u with gradient du we can change the point of evaluation via:

 © by the authors, 2017

C++ code

```
f.template update<0>( std::make_tuple(u,du) );
```

Similarly, for a perturbation v with gradient dv, we get:

C++ code

```
double value = f();
double df_dx = f.template d1<0>( std::make_tuple(v,dv) );
...
```

Note that different types for v and dv are required. Thus, this does not directly work for higher order ODE problems[17].

7 Conclusion

In this paper a library for the generation of invariant-based models and complex mathematical functions was presented. Exploiting features of C++11/14 a highly generic and still simple implementation is provided, admitting the computation of derivatives with respect general vector space elements, such as scalar, vector- or matrix-valued variables.

As building blocks for more complex functions FunG contains several mathematical functions from cmath as well as a large set of principal, mixed and modified matrix invariants. Moreover an arbitrary number – within the limits of the system architecture – of unknowns of different types can be used.

With the help of automatic type deduction complicated template-techniques are not exposed to users. This enables FunG to provide a small, intuitive and easily extensible interface. The optimization strategies of Sec. 4 as well as strict application of modern C++-techniques and SOLID principles, cf. Martin [2009], admit the generation of highly efficient code. This was demonstrated by the examples in Sec. 5.

[17]For the next version, there is an extension planned that admits to directly use gradient variables of the same type as the state.

© by the authors, 2017

8 Download and installation

Both, the current stable version (1.3.2) and the master branch can be accessed from

`http://lubkoll.github.io/FunG` or `https://github.com/lubkoll/FunG`.

Aiming at high quality, re-usable code, FunG was developed using continuous integration[18] and extensive unit testing with the Google C++ test framework, cf. Sen [2010][19].

FunG is licensed under the GNU General Public License v3.0 (GPL v3). For compilation a `C++`-compiler with support for the following features of `C++14` are required:

- `decltype(auto)` and
- the type traits aliases with trailing "`_t`".

For installation go to the root directory of FunG and issue the following commands[20]:

- `mkdir build`
- `cd build`
- `cmake ..`
- `make install`

To compile and run the unit tests issue the following commands in the build directory:

- `make`
- `ctest (--verbose)`

[18] `https://travis-ci.org/lubkoll/FunG`

[19] For test coverage see `https://coveralls.io/github/lubkoll/FunG`.

[20] Since FunG is header-only the `make install` command only copies the header files to the installation directory.

 © by the authors, 2017

References

SEMT. https://github.com/st-gille/semt.

A. Alexandrescu. Optimization Tips - Mo' Hustle Mo' Problems. https://www.youtube.com/watch?v=Qq_WaiwzOtI (CppCon 2014), 2014.

B. M. Bell. CppAD - A Package for Differentiation of C++ Algorithms. http://www.coin-or.org/CppAD.

C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report 1996-x5-94, Technical University of Denmark, 1996.

M. Blatt and P. Bastian. The Iterative Solver Template Library. *Appl. Par. Comp.*, 4699:666–675, 2007.

W. Brown. A SFINAE-Friendly std::common_type. Technical report, ISO/IEC JTC1/SC22/WG21, 2014.

J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *COOTS'98 Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, 1998.

A. Griewank and A. Walther. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

G. Guennebaud and B. Jacob. Eigen v3. http://eigen.tuxfamily.org, 2010.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

R. J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(26):1–16, 2014.

D. Kourounis, L. N. Gergidis, and M. A. Saunders. Compile-Time Symbolic Differentiation Using C++ Expression Templates.

L. Lubkoll. *An Optimal Control Approach to Implant Shape Design: Modeling, Analysis and Numerics.* PhD thesis, University of Bayreuth, 2015.

R. C. Martin. *Clean Code. A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

J. A. C. Martins, E. B. Pires, R. Salvado, and P. B. Dinis. A numerical model of passive and active behavior of skeletal muscles. *Comp. Meth. Appl. Mech. Eng.*, 151:419–433, 1998.

M. Sagebaum, T. Albring, and N. Gauger. CoDiPack - Code Differentiation Package. http://www.scicomp.uni-kl.de/software/codi/.

C. Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computational Intensive Experiments. Technical report, NICTA, 2010.

A. Sen. A quick introduction to the Google C++ Testing Framework. Technical report, IBM developerWorks, 2010.

G. Sommer, M. Eder, L. Kovacs, H. Pathak, L. Bonitz, C. Mueller, P. Regitnig, and G. A. Holzapfel. Multiaxial mechanical properties and constitutive modeling of human adipose tissue: A basis for preoperative simulations in plastic and reconstructive surgery. *Acta Biomater.*, 9:9036–9048, 2013.

© by the authors, 2017

R. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press (Free Software Foundation), 2015.

T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

A. Walther and A. Griewank. Getting started with ADOL-C. *Comb. Sci. Comp.*, pages 181–202, 2012.

 © by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 193–216
DOI: 10.11588/ans.2017.1.27720

Extending DUNE: The dune-xt modules

Tobias Leibner[1], René Milk*[1], and Felix Schindler[1]

[1]Institute for Computational and Applied Mathematics, University of Münster, Orleans-Ring 10, D-48149 Münster

Received: February 6th, 2016; **final revision:** October 11th, 2016; **published:** March 6th, 2017.

Abstract: We present our effort to extend and complement the core modules of the *Distributed and Unified Numerics Environment* DUNE (http://dune-project.org) by a well tested and structured collection of utilities and concepts. We describe key elements of our four modules dune-xt-common, dune-xt-grid, dune-xt-la and dune-xt-functions, which aim at further enabling the programming of generic algorithms within DUNE as well as adding an extra layer of usability and convenience.

1 Introduction

Over the past decades, numerical approximations of solutions of partial differential equations (PDEs) have become increasingly important throughout almost all areas of the natural sciences. At the same time, research efforts in the field of numerical analysis, computer science and approximation theory have led to adaptive algorithms to produce such approximations in an efficient and accurate manner: by now, for a large variety of PDEs advanced approximation techniques are available, such as wavelet methods [27], spectral methods [14], radial basis functions [8] and in particular grid-based discretization techniques such as Finite Volume and continuous or discontinuous Galerkin methods [10].

Focusing on the latter class, there exist a variety of open source and freely available PDE software frameworks with a strong scientific background, such as deal.ii [2], DUNE [4, 3], Feel++ [26], FreeFem++ [16], Fenics [20] and libMesh [19]. We consider the C++-based *Distributed and Unified Numerics Environment* (DUNE), which consists of the core modules dune-common, dune-geometry, dune-grid, dune-istl [6, 7] and dune-localfunctions[1], complemented by the discretization frameworks dune-fem [9, 12], dune-gdt[2] and dune-pdelab [5].

DUNE yields highly efficient programs, is fairly well documented and has a large developer and user base with a strong background in numerical analysis and scientific computing. However, since it is mainly targeted at researchers and centered around code efficiency, it has a steep learning curve and lacks some convenience features. In addition, despite enabling flexible numerical

*The author gratefully acknowledges funding by the DFG SPP 1648 "Software for Exascale Computing" under contract OH 98/5-1.

[1]All available under http://dune-project.org.

[2]https://github.com/dune-community/dune-gdt

schemes and providing an exceptionally powerful and generic grid concept, it does not allow for generic algorithms at all times, as detailed further below. To remedy this situation we propose a family of DUNE extension modules.

This paper is organized as follows: Section 2 gives a brief overview of `dune-xt` and presents our testing framework which is used in all modules. The remaining sections each give an in-depth discussion of one of the DUNE modules which make up `dune-xt`.

2 The `dune-xt` project

The `dune-xt` project aims to provide an extra layer of usability and convenience to existing DUNE modules and at the same time to enable programming of generic algorithms, in particular in the context of discretization frameworks. It consists of the four modules `dune-xt-common`, `dune-xt-grid`, `dune-xt-la` and `dune-xt-functions` (which we refer to as `dune-xt`), which are detailed throughout this paper.

We complement our discussion with extensive code listings and examples, which are not necessarily meant to be directly usable (for instance, we shall omit the `main()` in C++ code and frequently drop the `Dune::` namespace qualifier to improve readability). In addition we provide references to code locations, where "`dune/foo/bar.hh`" denotes the location of a file within the `dune-foo` module and "`dune/xt/foo/bar.hh`" denotes the location of a file within the `dune-xt-foo` module. However, we do not provide individual references for elements of the C++ standard template library, which are prefixed by `std::`.[3]

Since the main goal of `dune-xt` is to provide a solid infrastructure, it does not ship examples for the usage of all of its parts. Thus, we often refer to the generic discretization toolbox `dune-gdt`[4] for examples and motivation.

2.1 Code availability

The `dune-xt` modules are open source software and freely available on GitHub: https://github.com/dune-community/.[5] They are mainly developed by T. Leibner, R. Milk and F. Schindler with contributions from A. Buhr, S. Girke, S. Kaulmann, B. Verfürth and K. Weber, amounting to roughly 43.000 lines of code at the time of writing. All modules are dual licensed under a BSD 2-Clause License[6] or any GPL-2.0+ License[7] with linking exception[8], thus being license-compatible to DUNE. We refer to the supplementary material, which contains a snapshot of the `dune-xt-super`[9] module which bundles together all required dependencies and the `dune-xt` modules described in this paper.

2.2 Testing

Testing code and exposing it to as many different circumstances as possible is an extremely important part of software development, especially given the templated nature of DUNE and the large amount of provided grid managers. In order to actually take a burden off the developer, tests have to be executed automatically in a reliable manner on each published code change, should be carried out in as many different configurations as possible, and reports of test failure have to be delivered to the developer responsible for the change. In this section we describe the test procedure that is used in `dune-xt`.

[3] We refer to http://en.cppreference.com/ instead.
[4] https://github.com/dune-community/dune-gdt
[5] See Section 2.2 on why we cannot make use of https://gitlab.dune-project.org/.
[6] http://opensource.org/licenses/BSD-2-Clause
[7] http://opensource.org/licenses/gpl-license
[8] http://www.dune-project.org/license.html
[9] https://github.com/dune-community/dune-xt-super

 © by the authors, 2017

For testing, we use the Google C++ Testing Framework[10] (Gtest) in combination with the DUNE module dune-testtools[11], introduced in [18]. Further, we use Travis CI[12] to automatically run tests on each push to the code repository.

Gtest is a testing library for C++ code of the xUnit family [22]. It provides assertion macros that make it very easy to write tests. Assertions can be non-fatal, such that the test will continue to run and output failure information if an assertion fails. This allows for the detection of several faults in one test cycle. Gtest uses fixtures, "recipies" for parametrization, setup and teardown of tests, to make tests run independently of each other. Fixture classes allow to use the same configuration for several different tests and to share code between tests, minimizing the effort needed for writing and maintaining tests. Still, objects for each test are created only for that specific test and not shared between tests, which prevents hard-to-reproduce failures due to interacting tests.

We often want to run the same test for several related (template) classes. For this purpose, Gtest supports type parameterized test fixtures. The desired types have to be collected in the testing::Types struct and passed to a test macro that automatically runs the test for all types. However, constructing a large Types struct with permutations or products of type tuples is inconvenient. Moreover, Gtest currently only supports up to 50 types per test[13] which is frequently exceeded in dune-xt. Consider the following class from dune-xt-functions, which represents a scalar-, vector-, or matrix-valued constant function $f : \mathbb{R}^d \to \mathbb{R}^{r \times c}$

C++ code

```
1 template <class E, class D, size_t d, class R, size_t r, size_t rC>
2 class ConstantFunction;
```

and suppose we want to test the ConstantFunction class for all combinations of dimensions d, r, rC $\in \{1, 2, 3\}$ and domain and range field types D, R $\in$ {float, double}. This adds up to $2^2 \cdot 3^3 = 108$ different specialized ConstantFunction classes that would have to be manually written in the test code. Each time we want to add another field type or higher dimensions, a lot of specializations have to be manually added. In addition, macros are necessary to circumvent the 50 types limit of Gtest complicating the test code even more.

To circumvent these problems, we use the dune-testtools module which provides tools for system testing in DUNE. Among other features, dune-testtools makes it easy to create tests with dynamic and static variations. The configuration is done by providing a meta ini file that contains the possible variations (see Listing 1).

Code Listing 1: Meta ini file for test configuration

```
1  d  = 1, 2, 3 | expand
2  r  = 1, 2, 3 | expand
3  rC = 1, 2, 3 | expand
4  D = float, double | expand
5  R = float, double | expand
6  grid = SomeGridImplementation<{D}, {d}> | expand
7  E = {grid}::Codim<0>::Entity
8
9  [__static]
10 FUNCTIONTYPE = ConstantFunction<{E}, {D}, {d}, {R}, {r}, {rC}>
```

The expand command tells dune-testtools to create all possible values of the corresponding key, the curved brackets make dune-testtools paste the content of the embraced variable. FUNCTIONTYPE can be used like a preprocessor define in the associated C++ test code. Provided the meta ini file and the C++ test file, dune-testtools creates 108 executables which each test a single specialization of

[10] https://github.com/google/googletest
[11] https://gitlab.dune-project.org/quality/dune-testtools
[12] https://travis-ci.org/
[13] https://github.com/google/googletest/blob/master/googletest/include/gtest/internal/gtest-type-util.h

© by the authors, 2017

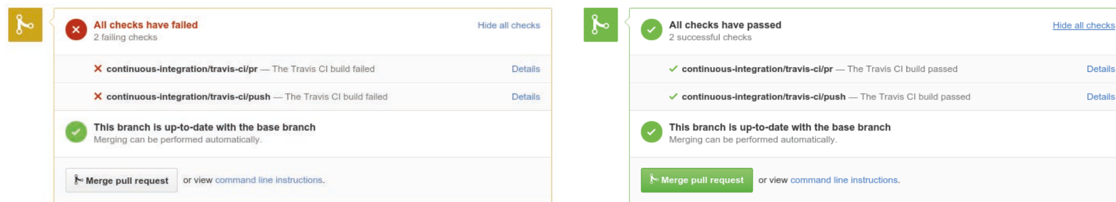


Figure 1: Travis integration with Github gives visual feedback for automatic tests (failing: left, succeeding: right), in particular to keep the workload for library developers at a minimum when handling code changes and contributions.

`ConstantFunction`. Adding another field type or higher dimensions is as simple as adding it to the meta ini file without modifying the test code at all. In addition, dynamic variations, i.e. parameters that are provided at runtime and not at compile time, can easily be added in a similar way, see [18].

Besides the reduced code complexity, this setup also makes it easier to debug tests as one can see at a glance which types are failing and debug only the associated tests. Creating an executable per tested type instead of testing all types in one executable using the `Types` struct greatly reduces memory consumption during compilation. As a downside we see increased total compile times. Depending on the available build infrastructure this can be offset by the greater available parallelism in building the test, resulting in more but smaller binaries.

Writing the tests is only the first step. Even the most comprehensive test suite is useless without getting regularly executed. We therefore continuously test each module on every update of the module's code repository using the Travis CI infrastructure (Travis)[14], whose use is free of charge for open source projects. Relying on Google's Compute Engine[15] and following the setup specified in the module's `.travis.yml` file Travis provides us with a highly flexible environment to run our test suites in. Using Travis' "build matrix"[16] feature our test suites are run with a diverse setup of available DUNE modules, from the minimal set of hard dependencies specified in each `dune.module` file to the full set of suggested ones, as well as different compilers (currently gcc 4.9 and 5.3 and clang 3.7). Travis offers tight integration with Github[17] pull requests (see Figure 1) and branches, each of which gets automatically tested whenever new commits get added, with visual on-page and email feedback. This is one of the main reasons why we host our modules on Github. In our opinion the prompt and automatic feedback on the validity of code contributions, with reasonably high coverage, is of enormous value to the development process.

3 The `dune-xt-common` module

Just like `dune-common`, we provide a common infrastructure in `dune-xt-common` with minimal dependencies, which is used throughout the other modules.

3.1 Improved handling of dense containers

`dune-common` contains the `FieldVector`[18] and `FieldMatrix`[19] classes, which model small dense vectors and matrices of fixed size (in particular used for coordinates, affine reference maps and function evaluations). These containers are implemented to provide maximum performance, but lack some convenience features. In particular the lack of certain operators and contructors make it difficult

[14] https://travis-ci.org/dune-community
[15] https://cloud.google.com/compute/
[16] https://docs.travis-ci.com/user/customizing-the-build/#Build-Matrix
[17] https://github.com
[18] dune/common/fvector.hh
[19] dune/common/fmatrix.hh

 © by the authors, 2017

to write generic code. Consider, for instance, a generic string conversion utility (assuming `T` is a matrix type):

C++ code

```
template <class T>
static inline T from_string(const std::string ss,
                            const size_t rows = 0,
                            const size_t cols = 0)
{
  T result(rows, cols); // <- does not compile for FieldMatrix
  // ... fill result from ss
  return result;
}
```

The above example does compile if `T` is a `DynamicMatrix`[20], but not if `T` is a `FieldMatrix`, which makes it extremely difficult to write generic code. And while it is clear that constructing a `FieldMatrix` of fixed size `M×N` is not sensible for other values of `rows` and `cols`, the construction in line 6 should be possible for `rows = M` and `cols = N` (throwing an appropriate exception otherwise).

In order to allow for generic algorithms we provide templated `VectorAbstraction` and `MatrixAbstraction` classes in `dune/xt/common/{matrix,vector}.hh` (along with specialization for all sensible vector and matrix classes), which allow for generic creation of and access to matrices and vectors. Additionally, we provide `is_vector` and `is_matrix` traits, which allow to rewrite the above example:

C++ code

```
#include <dune/xt/common/matrix.hh>
#include <dune/xt/common/type_traits.hh>

using namespace Dune::XT::Common;

template <class T>
  static inline typename std::enable_if<is_matrix<T>::value, T>::value
from_string(const std::string ss, const size_t rows = 0, const size_t cols = 0)
{
  auto result = MatrixAbstraction<T>::create(rows, cols);
  // ... fill result from ss using MatrixAbstraction< T >::set_entry(...)
  return result;
}
```

It is thus possible to use `from_string` with matrices of different type:

C++ code

```
#include <dune/common/fmatrix.hh>
#include <dune/common/dynmatrix.hh>

auto fmat = from_string<Dune::FieldMatrix<double, 2, 2>>("[1. 2.; 3. 4.]");
auto dmat = from_string<Dune::DynamicMatrix<double>>(    "[1. 2.; 3. 4.]");
```

In particular, one can use it with any custom matrix implementation by providing a specialization of `MatrixAbstraction` within the user code:

C++ code

```
class CustomMatrix { /* user provided matrix implementation */ };

struct MatrixAbstraction<CustomMatrix> { /* implement specialization */ };

auto cmat = from_string<CustomMatrix>("[1. 2.; 3. 4.]");
```

[20] `dune/common/dynmatrix.hh`

Based on these abstractions we provide many generic implementations in dune-xt. For instance, we provide an extension of the FloatCmp[21] mechanism from dune-common (see Section 3.4) for any combination of vectors (which allows for the same syntax as its counterpart in dune-common, see Section 3.4):

C++ code

```
#include <dune/common/dynvector.hh>
#include <dune/xt/common/float_cmp.hh>

std::vector< double >          svector({1., 1.});
Dune::DynamicVector< double > dvector(2, 1.);

Dune::XT::Common::FloatCmp::eq(svector, dvector);
```

3.2 Improved string handling and Configuration

As already hinted at above, we provide the string conversion utilities

C++ code

```
template< class T >
static inline T from_string(const std::string ss,
                            const size_t size = 0, const size_t cols = 0);

template< class T >
static inline std::string to_string(const T& ss);
```

in dune/xt/common/string.hh. These can be used with any basic type as well as with all matrices and vectors supported by the abstractions from Section 3.1. We use standard notation for vectors ("[1 2]") and matrices ("[1 2; 3 4]"), see the previous paragraph for examples. The from_string function takes optional arguments which determine the size of the resulting container (for containers of dynamic size), where 0 means automatic detection (the default).

Based on these string conversion utilities, we provide an extension of dune-common's ParameterTree[22] in dune/xt/common/configuration.hh: the Configuration class. The Configuration is derived from ParameterTree and can thus be used in all places where a ParameterTree is expected. While it also adds an additional layer of checks (in particular regarding provided defaults), report and serialization facilities, one of its main features is to allow the user to extract any type that is supported by the string conversion facilities. Given a sample configuration file in .ini format (for example the default configuration required to create a cubic grid using the factory methods discussed below),

Code Listing 2: Contents of the cube_gridprovider_default_config() in dune-xt-grid

```
type            = xt.grid.gridprovider.cube
lower_left      = [0 0 0 0]
upper_right     = [1 1 1 1]
num_elements    = [8 8 8 8]
num_refinements = 0
overlap         = [1 1 1 1]
```

we can query the resulting Configuration object config for the types supported by the ParameterTree,

C++ code

```
auto num_refinements = config.get<int>("num_refinements");
```

as well as for all types supported by our string conversion utilities (including custom matrix and vector types as explained in the previous paragraph):

[21] dune/common/float_cmp.hh
[22] dune/common/parametertree.hh

 © by the authors, 2017

C++ code

```
auto lower_left = config.get<FieldVector<double, dimDomain>>("lower_left");
```

The above is valid for all $0 \leq$ `dimDomain` ≤ 4, due to the automatic size detection of `from_string`.

3.3 Timings

Getting information about how long certain portions of an application take to run is crucial to guide optimization efforts. It can also be useful to estimate the entire loop execution time from the first couple of loops and of course for benchmarking parts of or an entire application for different parameters. Mature and generally simple to use tools exist, both commercial (e.g., Intel VTune[23], Allinea Map[24]) and free (e.g., Vagrind/Callgrind[25], Oprofile[26]), that allow very fine-grained performance analysis of a given code, down to function or even instruction level. It is however very hard or impossible to measure custom, arbitrary sections of code with these tools and to present those measurements in an easily understood format. The `dune-common` module provides a straightforward implementation of a `Timer`[27] class that relies on `std::system_clock` to measure expired walltime. This is enough for simple needs, but we extend on this concept with the `Timings` and related classes in `dune/xt/common/timings.hh`, because measuring user and system time can be greatly insightful and having a centralized managing facility with its own output capabilities is much easier for the user. The global `Timings` instance keeps a map of named sections and associated `TimingData` objects. The user can start/stop sections manually or use a guard object to start a timing at object construction and stop it when the guard object goes out of scope:

Code Listing 3: Timing example

```
using namespace Dune::XT::Common;
timings().start("sec");
for (auto i : value_range(5)) {
  ScopedTiming scoped_timing("sec.inner");
  busywait(i * 200);
}
timings().stop("sec");
busywait(1000);
auto file = make_ofstream("example.csv");
timings().output_all_measures(*file);
```

Internally `TimingData` expands on `Timer` by measuring wall, system and user time between start and stop concurrently using a `boost::timer::cpu_timer`[28]. `Timings` has a couple of `output_*` functions that all write data in the Comma Separated Values (CSV) format and which distinguish themselves from one another in which measures are produced and whether averages, minima and maxima are calculated over MPI-ranks. We chose the CSV format for our output because it enables post processing and analysis by a wide variety of tools, e.g. Pandas [21] or matplotlib [17].

Code Listing 4: example.csv

```
threads,ranks,sec_avg_usr,sec_max_usr,sec_avg_wall,sec_max_wall,
    sec_avg_sys,sec_max_sys,sec.inner_avg_usr,sec.inner_max_usr,
    sec.inner_avg_wall,sec.inner_max_wall,sec.inner_avg_sys, sec.inner_max_sys
8,1,2050,2050,2050,2050,0,0,2050,2050,2050,2050,0,0
```

[23] https://software.intel.com/en-us/intel-vtune-amplifier-xe
[24] http://www.allinea.com/products/map
[25] http://valgrind.org/info/tools.html#callgrind
[26] http://oprofile.sourceforge.net/
[27] dune/common/timer.hh
[28] http://www.boost.org/doc/libs/1_58_0/libs/timer/doc/cpu_timers.html#Class-cpu_timer

© by the authors, 2017

3.4 Extending Float Comparison

Floating point operations are not always exact, so using `operator==` to compare primitive floating point types may yield unwanted results. To avoid this problem, `dune-common` provides functions `eq`, `ne`, `gt`, `lt`, `ge` and `le` in the namespace `Dune::FloatCmp` to approximately compare floating point numbers. There are three variants of approximate comparisons (`absolute`, `relativeWeak` and `relativeStrong`) implemented in `dune-common`. The `absolute` compare style checks two floating point numbers a and b for equality by

$$|a - b| \leq \epsilon_{\text{abs}},$$

where ϵ_{abs} is a specified tolerance parameter. This works well for numbers that are neither too small nor too large, but two numbers with absolute values lesser than ϵ_{abs} will always be specified as equal while for large numbers the gap between two adjacent floats may become greater than ϵ_{abs} rendering the result the same as with `==`. The compare styles `relativeWeak`

$$|a - b| \leq \epsilon_{\text{rel}} \max(|a|, |b|)$$

and `relativeStrong`

$$|a - b| \leq \epsilon_{\text{rel}} \min(|a|, |b|)$$

use a tolerance parameter that is scaled with the numbers and thus are suitable for a wide range of numbers. However, problems occur if one of the numbers is zero. Thus, we provide a fourth compare style `numpy` in `dune/xt/common/float_cmp.hh` that is equivalent to the implementation in `numpy.isclose`[29]. Here, both an absolute tolerance ϵ_{abs} and a relative tolerance ϵ_{rel} are chosen and the numbers a, b are considered equal if

$$|a - b| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} |b| .$$

This allows for an absolute comparison (with a sufficiently small ϵ_{abs}) near 0 and a relative comparison elsewhere and is thus the default compare style in `dune-xt-common`. Note that this comparison is not symmetric with respect to a, b, so `XT::Common::FloatCmp::eq(a,b)` may not be the same as `XT::Common::FloatCmp::eq(b,a)` in some cases. The `absolute`, `relativeWeak` and `relativeStrong` compare styles are also supported by `dune-xt-common` and can be chosen by providing a template parameter to the comparison functions.

Often we rather want to compare vectors of floating point numbers instead of scalar values. This could be done by looping over the components of the vectors and comparing them one by one, which is inconvenient for the user. `dune-xt-common` has built-in support for vector comparisons, i.e. one can compare any two vectors as long as there are `VectorAbstractions` (see Section 3.1) available for both vector types. In particular, `std::complex` and vectors containing `std::complex` values are supported.

4 The `dune-xt-grid` module

The `dune-xt-grid` module builds on `dune-grid` and `dune-xt-common` to provide powerful concepts to improve performance (such as the `EntityInlevelSearch` and the `Walker`) or to allow for generic algorithms (such as the `GridProvider`).

4.1 Searching the grid

A typical task within a PDE solver is to prolong discrete function data from one discrete function space onto another one (that is usually associated with a finer grid). Without a direct, one-to-one identification mapping of degrees of freedom (DoF) available (think of different discrete functions stemming from `dune-fem` and `dune-pdelab`), this requires identifying those entities in the source space where the source function should be evaluated in quadrature points defined

[29] http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.isclose.html

 © by the authors, 2017

with respect to the range space. With the EntityInlevelSearch we provide a means to that end in dune/xt/grid/search.hh for arbitrary grids, as long as the range grid covers a subset of the physical domain of the source grid.

C++ code

```
template <class GridViewType, int codim = 0>
class EntityInlevelSearch : public EntitySearchBase<GridViewType>
{
  // not all types shown
public:
  EntityInlevelSearch(const GridViewType& gridview)
    : gridview_(gridview)
    , it_last_(gridview_.template begin<codim>())
  {}

  template <class PointContainerType>
  EntityVectorType operator()(const PointContainerType& points);

private:
  const GridViewType gridview_;
  IteratorType it_last_;
};
```

It is modeled after the HierarchicSearch[30] in dune-grid, with the extension that instead of finding one Entity for one point at a time we allow searching for a sequence of points (via operator()) and return a sequence of entities accordingly. As another improvement the implementation remembers the last search position on the source grid across search calls. This greatly improves the performance over the naive implementation of restarting the search for every operator() in case of equivalent grids.

4.2 Providing generic access to grid views and grid parts

There exists an unfortunate disagreement between dune-grid and dune-fem, whether grid views or grid parts are to be used to model a collection of grid elements, which makes it hard to implement generic algorithms. Think of some code which needs to create a LevelGridView or LevelGridPart, depending on the further use for a discrete function space implemented via dune-pdelab or dune-fem:

C++ code

```
#include <dune/fem/gridpart/levelgridpart.hh>

template <class GridType>
void create_level(GridType& grid, const int lv) {
  // either ?
  auto level_view = grid.levelGridView(lv);
  // or ?
  Dune::Fem::LevelGridPart<GridType> level_part(grid, lv);
  // ...
}
```

In dune/xt/grid/gridprovider.hh we thus provide the GridProvider, the purpose of which is to hold a the grid object and to allow for a generic creation of grid views and grid parts to ultimately allow for generic code despite the above mentioned disagreement:

C++ code

```
template <class GridType>
class GridProvider
{
  // not all methods and types shown ...
```

[30] dune/grid/utility/hierarchicsearch.hh

```
5  public:
6    std::shared_ptr<GridType>& grid_ptr();
7    GridType& grid();
8
9    template <Layers layer_type, Backends backend>
10   typename Layer<layer_type, backend>::Type layer(const int lv = 0);
11 };
```

Together with the `Layers` and `Backends` enum classes[31], this allows to write generic code by passing a grid provider along with the required tag:

C++ code

```
1  #include <dune/xt/grid/provider.hh>
2
3  using namespace Dune::XT::Grid;
4
5  template <class GridType, Backends backend>
6  void create_level(GridProvider<GridType>& grid_provider, const int lv) {
7    auto level_part_or_view = grid_provider.layer<Layers::level, backend>(lv);
8    // ...
9  }
```

We provide several factory methods to create a `GridProvider` (for instance `make_cube_grid`, which creates grids of rectangular domains and `make_dgf_grid` and `make_gmsh_grid`, which either create grids from the respective definition file). In addition, we provide a means to select a factory at runtime using the `GridProviderFactory` struct, given the type of the grid `G` and a `Configuration` object `config` (for instance using the default one for `make_cube_grid`, see Listing 2),

C++ code

```
1  #include <dune/xt/gridprovider.hh>
2
3  #using namespace Dune::XT::Grid;
4
5  auto grid_provider = GridProviderFactory<G>::create("xt.grid.gridprovider.cube",
6                                                       config);
```

which is particularly useful in conjunction with configuration files or *Python* bindings.

4.3 Identification of domain boundaries

PDE solvers need to prescribe the solution's behaviour on the boundary of the computational domain. In general, there may exist a large number of different boundary conditions in any given problem, and therefore any PDE solver needs to provide a way to identify parts of that domain boundary and categorize them. Unfortunately, `dune-grid` does not provide such a mechanism.

This problem is tackled in `dune-pdelab`, for instance, by deriving from `Dune::TypeTree::LeafNode`[32] and implementing the following methods:

C++ code

```
1  virtual bool isDirichlet(const IntersectionGeometryType& intersection_geometry,
2                           const FieldVector<D, d - 1>& coord) const;
3
4  virtual bool isNeumann(const IntersectionGeometryType& intersection_geometry,
5                         const FieldVector<D, d - 1>& coord) const;
```

[31] `dune/xt/grid/layers.hh`
[32] `dune/typetree/leafnode.hh`

 © by the authors, 2017

This approach is however limited to the boundary types defined in dune-pdelab and the user has no means to extend this approach to other boundary type (one cannot just add a isCustomBoundary() method to the interface).

We therefore propose a more flexible mechanism that is based on mapping the boundary category to types derived from BoundaryType[33].

C++ code

```
class BoundaryType
{
protected:
  virtual std::string id() const = 0;

public:
  virtual bool operator==(const BoundaryType& other) const
  {
    return id() == other.id();
  }
};

class DirichletBoundary : public BoundaryType
{
protected:
  virtual std::string id() const override final
  {
    return "dirichlet boundary";
  }
};

class NeumannBoundary : public BoundaryType { /*...*/};

class RobinBoundary : public BoundaryType { /*...*/};
```

For a given grid view (and its type of intersection), access to the type of the boundary is granted by the BoundaryInfo class concept

C++ code

```
template <class IntersectionType>
class BoundaryInfo
{
public:
  virtual BoundaryType& type(const IntersectionType& intersection) const = 0;
};
```

based on which a PDE solver library (such as dune-gdt) can provide generic algorithms which act only on parts of the domain boundary by checking

C++ code

```
if (boundary_info.type(intersection) == DirichletBoundary()) {
  // ...
} else if (boundary_info.type(intersection) == NeumannBoundary()) {
  // ....
} else
  // ...
```

In addition, this concept can be easily extended by the user by simply deriving from BoundaryType. As implementations of the BoundaryInfo concept, we currently provide:

- AllDirichlet and AllNeumann, the purpose of which is self-explanatory.

[33] dune/xt/grid/boundaryinfo/types.hh

© by the authors, 2017

- `NormalBased`, which allows to identify boundary intersections by the direction of their outward pointing normal.

In order to allow for problem definition classes to define domain boundaries independently of the type of the grid, we also provide the `BoundaryInfoFactory`. User classes can hold a complete description of one of the boundary infos from above in a `Configuration` instance. Given the type of the grid (and thus the type of an intersection), one is then able to create an instance of one of the implementations of the `BoundaryInfo` concept of correct type, as required:

C++ code

```
#include <dune/grid/yaspgrid.hh>
#include <dune/grid/sgrid.hh>
#include <dune/xt/common/configuration.hh>
#include <dune/xt/grid/boundaryinfo.hh>

using namespace Dune::XT::Common;
using namespace Dune::XT::Grid;

class Problem
{
public:
  Configuration boundary_info_cfg()
  {
    Configuration config;
    config["type"]       = "xt.grid.boundaryinfo.normalbased";
    config["default"]    = "dirichlet";
    config["neumann.0"] = "[ 1. 0.]";
    config["neumann.1"] = "[-1. 0.]";
    return config;
  }
};

typedef typename Dune::YaspGrid<2>::LeafIntersection YI;
typedef typename Dune::SGrid<2, 2>::LeafIntersection SI;

Problem problem;
auto boundary_info_y = BoundaryInfoFactory<YI>::create(problem.boundary_info_cfg());
auto boundary_info_s = BoundaryInfoFactory<SI>::create(problem.boundary_info_cfg());
```

The type of `boundary_info_y`, for instance, is `std::unique_ptr<BoundaryInfo<YI>>`. Given a rectangular domain in $\mathbb{R}^2$, it models a Neumann boundary left and right and a Dirichlet boundary everywhere else.

4.4 Periodic Gridviews

Periodic boundary conditions are frequently used to model an infinite domain. To apply periodic boundary conditions in the context of Finite Volume methods, intersections on the periodic boundary have to be linked to the intersection on the opposite side of the (finite) grid. For Finite Element methods, the indices of connected degrees of freedom (DoFs) on the boundary have to be identified and suitable constraints applied. In any case, the implementation can be quite time-consuming and cumbersome if there is no support from the underlying grid manager.

Support for periodic boundary conditions in DUNE is varying from grid to grid. `YaspGrid` supports periodic boundaries using the parallel DUNE grid interface which requires the user to run a parallel program. Several grid managers including **dune-alugrid** [1] and SPGrid [25] support periodicity by gluing together edges of the unit cube [25]. While this allows to obtain the neighboring entity of an intersection on the periodic boundary, DoFs still have to be modified by hand.

Regarding the uneven support for periodic boundary conditions in DUNE, we provide the `PeriodicGridView` class in `dune/xt/grid/periodic_gridview.hh` which is derived from a given arbitrary `Dune::GridView`. As long as the `GridView` models an axis-parallel hyperrectangle with conforming

 © by the authors, 2017

faces, it is enough to replace a GridView by the corresponding PeriodicGridView in existing code to apply periodic boundary conditions (see Listing 5). This is internally achieved by replacing the IndexSet and iterators and the corresponding methods (begin, end, ibegin, iend, size and indexSet) of the GridView by periodic variants, utilizing the search capabilities from Section 4.1. The remaining methods are forwarded to the underlying GridView. Periodic directions can be specified by supplying a std::bitset (see Listing 5). By default, all directions are made periodic.

Code Listing 5: Usage of PeriodicGridView

```
using namespace Dune::XT::Grid;
GridViewType grid_view = grid.leafGridView();
std::bitset<3> periodic_directions(std::string("100")); // periodic in z-direction
PeriodicGridView<GridViewType> periodic_grid_view(grid_view, periodic_directions);
// use periodic_grid_view from here on to apply periodic boundary conditions ...
```

The PeriodicIntersectionIterator that can be obtained by the ibegin and iend methods returns a PeriodicIntersection when dereferenced. The PeriodicIntersection behaves exactly like the non-periodic Intersection except that it returns neighbor()== true and an outside() entity on the periodic boundary. The outside() entity is the entity adjacent to the periodically equivalent intersection, i.e., the intersection at the same position on the opposite side of the domain. The indexSet() method returns a PeriodicIndexSet which assigns the same index to entities that are periodically equivalent. Hence, the PeriodicIndexSet is usually smaller than the corresponding IndexSet. The begin and end methods of PeriodicGridView return a PeriodicIterator which visits only one of several periodically equivalent entities.

Note that there existed a similar but independently developed class PeriodicGridPart in dune-fem that contained a periodic index set but no periodic intersections. Unfortunately, the PeriodicGridPart was removed in the dune-fem release 1.2..

4.5 Walking the grid

Since we are considering grid-based numerical methods, we frequently need to iterate over the elements $t \in \tau_h$ of a grid view. In order to minimize the amount of required grid iterations, we want to be able to carry out N operations on each grid element, as opposed to walking the entire grid N times. We thus provide in dune/xt/grid/walker.hh the templated Walker class, working with any GridView (from dune-grid) or GridPart (from dune-fem):

C++ code

```
template <class GridViewType>
class Walker
{
  // not all methods and types shown ...
public:
  void add(Functor::Codim0<GridViewType>& functor/*, ... */);
  void add(Functor::Codim1<GridViewType>& functor/*, ... */);
  void add(Functor::Codim0And1<GridViewType>& functor/*, ... */);

  void walk(const bool use_tbb = false);
};
```

The user can add an arbitrary amount of functors to the Walker, all of which are then locally executed on each grid element. Each functor is derived from one of the virtual interfaces Functor::Codim0, Functor::Codim1 or Functor::Codim0And1, for instance

C++ code

```
template <class GridViewType>
class Codim0
{
public:
  typedef typename XT::Grid::Entity<GridViewType>::Type EntityType;
```

```
6
7     virtual void prepare() {}
8     virtual void apply_local(const EntityType& entity) = 0;
9     virtual void finalize() {}
10  };
```

where `prepare` (and `finalize`) are called before (and after) iterating over the grid, while `apply_local` is called on each entity of the grid. The user might note here the usage of traits from `dune/grid/{entity,intersection}.hh` to extract required information from a `GridView` or `GridPart` in a generic way. Each `add` method of the `Walker` accepts an additional argument which allows to select the elements and faces the functor will be applied on. For instance, in the context of a discontinuous Galerkin discretization in `dune-gdt` we want to apply local coupling operators on all inner faces of the grid and local boundary operators on all Dirichlet faces of the grid. Presuming we were given suitable implementations of these local operators as functors and a `BoundaryInfo` object in the sense of the previous paragraph, the following would realize just that:

C++ code

```
1  #include <dune/xt/grid/walker.hh>
2
3  using namespace Dune::XT::Grid;
4
5  Walker<GV> walker(grid_view);
6  walker.add(coupling_operator, new ApplyOn::InnerIntersectionsPrimally<GV>());
7  walker.add(boundary_operator, new ApplyOn::DirichletIntersections<GV>(boundary_info));
8  // add more, if required...
9  walker.walk()
```

The use of the `new` keyword will not create memory leaks here as the pointer is wrapped into a `std::unique_ptr` in the `add` method of the walker. Note that the `walk` method allows to switch between a serial and a shared memory parallel iteration over the grid, at runtime (via the `use_tbb` switch). In particular, the user only has to provide implementations of the functors and need not deal with any parallelization issues (or different types of grid walkers, depending on the parallelization paradigm).

5 The `dune-xt-la` module

As discussed (Section 3.1), `dune-common` provides small dense vectors and matrices which are, e.g., used for coordinates. In addition, any PDE solver requires large vectors and (usually sparse) matrices to represent assembled functionals and operators stemming from the underlying PDE. Linear algebra containers are a performance critical aspect of any discretization framework and one usually does not want to bet on a single horse: there exists external backends which are well suited for serial and shared memory parallel computations (such as EIGEN[15]) while others are more suited for distributed memory parallel computations (such as `dune-istl`). Neither is fitting for every purpose and we thus require a means to exchange the implementation of matrices and vectors depending on the circumstances. This calls for abstract interfaces for containers and linear solvers, which we provide within the `dune-xt-la` module.

5.1 Generic linear algebra containers

We chose a combination of static and dynamic inheritance for these interfaces, allowing for virtual function calls that act on the whole container (such as `scal`) while using the “Curiously recurring template patterns” (CRTP, see [11]) paradigm for methods that are called frequently (such as access to individual elements in loops), allowing the compiler to optimize performance critical calls (for instance by inlining). We provide a thread-safe helper class `CRTPInterface` in `dune/xt/common/crtp.hh` along with thread-safe `CHECK_...` macros for debugging, as the tools provided in `dune/common/bartonnackmanifcheck.hh` may not work properly in a shared memory parallel program.

 © by the authors, 2017

All matrices and vectors are derived from `ContainerInterface`[34]:

C++ code

```
template <class Traits, class ScalarType = typename Traits::ScalarType>
class ContainerInterface
  : public CRTPInterface<ContainerInterface<Traits, ScalarType>, Traits>
{
  // not all methods and types shown ...
public:
  typedef typename Traits::derived_type derived_type;

  // Sample CRTP implementation:
  inline void scal(const ScalarType& alpha)
  {
    // as_imp() from CRTPInterface performs a static_cast into derived_type.
    // Thus, scal() of the derived class is called.
    CHECK_AND_CALL_CRTP(this->as_imp().scal(alpha));
  }
  inline void axpy(const ScalarType& alpha, const derived_type& xx);
  inline derived_type copy() const;

  // Default implementation:
  // could be overriden by any derived class.
  virtual derived_type& operator*=(const ScalarType& alpha)
  {
    scal(alpha);
    return this->as_imp();
  }
};
```

The `ContainerInterface` enforces just enough functionality to assemble a linear combination of matrices or vectors, which is frequently required in the context of model reduction (compare [24]). Given an affine decomposition of a parametric matrix or vector B, we need to assemble $\sum_{q=0}^{Q-1} \theta_q B_q$ for given component containers B_q and scalar coefficients θ_q. The following generic code will work for any matrix or vector type `C` derived from `ContainerInterface`:[35]

```
#include <dune/xt/la/container/container-interface.hh>

using namespace Dune::XT::LA;

template <class C>
    typename std::enable_if<is_container<C>::value, C>::type
assemble_lincomb(const std::vector<C>& components,
                 const std::vector<double>& coefficients)
{
  auto result = components[0].copy();
  result *= coefficients[0];
  for (size_t qq = 1; qq < components.size(); ++qq)
    result.axpy(coefficients[qq], components[qq]);
  return result;
}
```

All containers in dune-xt-la are implemented with “copy-on-write” (COW) [23, 190-194]. Implementing data sharing among entities with data duplication only occurring if an entity tries to modify the referenced data is a well established and common technique in Computer Science. This pattern (also sometimes denoted as “lazy copy”) typically incurs minimal runtime overhead for the required reference counting for the great benefit of being able to pass around copies of an entity without immediate expensive memory copies. Since all container implementations in dune-xt-la are proxy classes that forward operations to the respective backend instances, we

[34] `dune/xt/la/container/container-interface.hh`

[35] Note the use of the `is_container` traits in line 6 that we provide in `dune/la/container/container-interface.hh`. Together with `std::enable_if`, this checks that `C` is derived from `ContainerInterface` at compile time.

© by the authors, 2017

insert the COW logic into the `backend` call, which is then used throughout the class to access the backend (in addition to allowing the user to directly access the underlying backend):

C++ code

```
1 BackendType& backend()
2 {
3   ensure_uniqueness();
4   return *backend_;
5 }
```

The `backend_` is simply a `std::shared_ptr<BackendType>` on which the `ensure_uniqueness` method can query the current status by a call to `unique` and perform the deep copy, if required. Note that due to COW and move semantics for all matrices and vectors in `dune-xt-la`, the only deep copy in Listing 5.1 is actually done in line 11 (neither in line 10 nor in line 14)

Based on `ContainerInterface` we provide the `VectorInterface`[36] for dense vectors and the `MatrixInterface`[37] for dense and sparse matrices. Each derived vector class has to implement the methods `size`, `add_to_entry`, `set_entry` and `get_entry_ref`, which allow to access and change individual entries of the vector. The interface provides default implementations for all relevant mathematical operators, support for range-based for loops and many useful methods, such as `dot`, `mean`, `standard_deviation` and `l2_norm`, just to name a few.

Each derived matrix class has to implement `rows`, `cols`, `add_to_entry`, `set_entry` and `get_entry` to allow for access to individual entries, `mv` for matrix/vector multiplication and `clear_row`, `clear_col`, `unit_row` and `unit_col`, which are required in the context of solving PDEs with Dirichlet Constraints or pure Neumann problems. Every matrix implementation (even a dense one) is constructible from the same type of sparsity pattern (which we provide in `dune/xt/la/container/pattern.hh`) and the access methods `..._entry` are only required to work on entries that are contained in the pattern. The interface provides several mathematical operators, norms and a means to obtained a pruned matrix (where all entries close to zero are removed from the pattern).

We also provide several vector and matrix implementations:

- The `CommonDenseVector` and `CommonDenseMatrix` in `dune/xt/la/container/common.hh`, based on the `DynamicVector` and `DynamicMatrix` from `dune-common`. These are always available.
- The `EigenDenseVector`, `EigenMappedDenseVector`, `EigenDenseMatrix` and `EigenRowMajorSparseMatrix` in `dune/xt/la/container/eigen.hh`, based on the `EIGEN` package (if available). The `EigenMappedDenseVector` allows to wrap an existing `double*` array and the `EigenRowMajorSparseMatrix` allows to wrap existing matrices in standard CSR format, which allows to wrap other container implementation (for instance in the context of *Python* bindings).
- The `IstlDenseVector` and `IstlRowMajorSparseMatrix` in `dune/xt/la/container/istl.hh`, based on `dune-istl` (if available).

To allow for generic algorithms we also provide the `Backends` enum class along with the `default_backend`, `default_sparse_backend` and `default_dense_backend` defines (which are set depending on the build configuration). These can be used in other libraries and user code to obtain suitable containers independently of the current build configuration.

Consider, for instance, an L^2 projection of a function $f : \Omega \to \mathbb{R}$ onto a discontinuous Galerkin discrete function space defined on a grid view τ_h modeling Ω. Given local basis functions φ_i^t,

[36] `dune/xt/la/container/vector-interface.hh`
[37] `dune/xt/la/container/matrix-interface.hh`

 © by the authors, 2017

$0 \leq i < I$ of such a space on each entity $t \in \tau_h$, the local DoF vector $\underline{f_h^t} \in \mathbb{R}^I$ of the projected discrete function is given as the solution of the linear system

$$\underline{L^{2^t}} \cdot \underline{f_h^t} = \underline{l_h^t}, \qquad \text{with} \qquad \left(\underline{L^{2^t}}\right)_{i,j} := \int_t \varphi_i^t \varphi_j^t \qquad \text{and} \qquad \left(\underline{l_h^t}\right)_i := \int_t f \varphi_i^t. \tag{1}$$

The problem of assembling and solving (1) is a typical situation within any PDE solver. In dune-gdt, for instance, (1) is roughly assembled as follows, given a local basis and an appropriate quadrature:

C++ code

```cpp
#include <dune/xt/la/container.hh>

using namespace Dune::XT::LA;

typedef typename Container<double, default_dense_backend>::MatrixType LocalMatrixType;
typedef typename Container<double, default_dense_backend>::VectorType LocalVectorType;

// ... on each grid element
LocalMatrixType local_matrix(local_basis.size(), local_basis.size(), 0.);
LocalVectorType local_vector(local_basis.size(), 0.);
LocalVectorType local_DoFs(local_basis.size(), 0.);

// ... at each quadrature point, given evauations of the local basis as basis_values
//     and of the source function as source_value
for (size_t ii = 0; ii < local_basis.size(); ++ii) {
  local_vector[ii] += integration_element * quadrature_weight
                    * (source_value * basis_values[ii]);
  for (size_t jj = 0; jj < local_basis.size(); ++jj) {
    local_matrix.add_to_entry(ii, jj,
                              integration_element * quadrature_weight
                              * (basis_values[ii] * basis_values[jj]));
  }
}
```

Note that we neither have to manually specify the correctly matching matrix and vector types nor to include the correct headers (which depend on the current build configuration). The `Container` traits together with any of the `default_...` defines always yields appropriate available types (lines 1, 5, 6).

5.2 Runtime selectable solvers

In order to determine the local DoF vector in the above example, we need to solve the algebraic problem: find `local_DoFs`, such that

$$\texttt{local_matrix} \cdot \texttt{local_DoFs} = \texttt{local_vector}.$$

In addition to such small, dense problems we also require the inversion of large (sparse) system and product matrices. For interesting large and real-world problems, however, there are few linear solvers available which can be used as a black box (if at all). Most problems require a careful choice and detailed configuration of the correct linear solver. We thus require access to linear solvers which can be used in a generic way but also exchanged and configured at runtime.

In `dune/xt/la/solver.hh` we provide such solvers via the `Solver` class:

C++ code

```cpp
template <class MatrixType>
class Solver
{
  // simplified variant
public:
```

```
6     Solver(const MatrixType& matrix);
7
8     static std::vector<std::string> types();
9
10    static Configuration options(const std::string type = "");
11
12    template <class RhsType, class SolutionType>
13    void apply(const RhsType& rhs, SolutionType& solution) const;
14
15    template <class RhsType, class SolutionType>
16    void apply(const RhsType& rhs, SolutionType& solution,
17               const std::string& type) const;
18
19    template <class RhsType, class SolutionType>
20    void apply(const RhsType& rhs, SolutionType& solution,
21               const Configuration& options) const;
22  };
```

We provide specializations of the `Solver` class for all matrix implementations derived from `MatrixInterface` (see the previous paragraph). To continue the example from the previous paragraph, this allows to determine the local DoF vector of the L^2 projection:

C++ code

```
1   #include <dune/xt/common/exceptions.hh>
2   #include <dune/xt/la/solver.hh>
3
4   try {
5     XT::LA::Solver<LocalMatrixType>(local_matrix).apply(local_vector, local_DoFs);
6   } catch (XT::Exceptions::linear_solver_failed& ee) {
7     DUNE_THROW(Exceptions::projection_error,
8                "L2 projection failed because a local matrix could not be inverted!\n\n"
9                << "This was the original error: " << ee.what());
10  }
```

The above example shows a typical situation within the library code of `dune-gdt`: we need to solve a small dense system for provided matrices and vectors of unknown type. We can do so by instantiating a `Solver` and calling the black-box variant of `apply` (line 5). This apply variant is default implemented by calling

C++ code

```
1   apply(rhs, solution, types()[0]);
```

where `types()` always returns a (non-empty) list of available linear solvers for the given matrix type, in descending priority (meaning the first is supposed to "work best"). For instance, if `local_matrix` was an `EigenDenseMatrix`, a call to `types()` would reveal the following available linear solvers:

```
1   {"lu.partialpiv", "qr.householder", "llt", "ldlt", "qr.colpivhouseholder",
2    "qr.fullpivhouseholder", "lu.fullpiv"}
```

On the other hand, if the matrix was an `EigenRowMajorSparseMatrix`, a call to `types()` would yield

```
1   {"bicgstab.ilut", "lu.sparse", "llt.simplicial", "ldlt.simplicial",
2    "bicgstab.diagonal", "bicgstab.identity", "qr.sparse",
3    "cg.diagonal.lower", "cg.diagonal.upper", "cg.identity.lower",
4    "cg.identity.upper"}
```

Given a (large sparse) `system_matrix` (for instance stemming from a discretized Laplace operator) and `rhs` vector, we can solve the corresponding linear system using a specific solver by calling

C++ code

```
1   #include <dune/xt/la/solver.hh>
2
```

 © by the authors, 2017

```
3 XT::LA::Solver<SystemMatrixType> linear_solver(system_matrix);
4 linear_solver.apply(rhs, solution, "ldlt.simplicial");
```

The above call to `apply` is default implemented by calling

C++ code

```
1 apply(rhs, solution, options(type));
```

where `options(type)` always returns a `Configuration` object with appropriate options for the selected `type`. With `type = "ldlt.simplicial"`, for instance, we are implicitly using the following options (which are the default for `"ldlt.simplicial"`):

```
1 type                       = ldlt.simplicial
2 post_check_solves_system = 1e-5
3 check_for_inf_nan          = 1
4 pre_check_symmetry         = 1e-8
```

All implemented solvers default to checking whether the computed solution does actually solve the linear system and provide additional sanity checks. We make extensive use of exceptions if any check is violated, which allows to recover from undesirable situations in library code (as shown in the example above). We also provide our own implementation of the `DUNE_THROW` macro in `dune/xt/common/exceptions.hh` which replaces the macro from `dune-common` and can be used with any `Dune::Exception`. Apart from a different formatting (and colorized output), it also provides information of interest in distributed memory parallel computations.

The `"ldlt.simplicial"` solver, for instance, does only work for symmetric matrices and we thus check the matrix for symmetry beforehand (which can be disabled by setting `"pre_check_symmetry"` to `0`). Each type of linear solver provides its own options, which allow the user to fine-tune the linear solver to his needs. For instance, the iterative `"bicgstab"` solver with `"ilut"` preconditioning accepts the following options (in addition to `"post_check_solves_system"` and `"check_for_inf_nan"`, which are always supported):

C++ code

```
1  #include <dune/xt/la/solver.hh>
2
3  XT::LA::Solver<SystemMatrixType> linear_solver(system_matrix);
4  auto options = linear_solver.options("bicgstab.ilut");
5
6  options["max_iter"]   = 1000;
7  options["precision"] = "1e-14";
8  options["preconditioner.fill_factor"] = 10;
9  options["preconditioner.drop_tol"]     = "1e-4";
10
11 linear_solver.apply(rhs, solution, options);
```

The actually implemented variant of `Solver` in `dune-xt-la` also takes a communicator as an optional argument, to allow for distributed parallel linear solvers. We provide several implementations of such parallel solvers based on `dune-istl`. A linear solver for `IstlRowMajorSparseMatrix`, for instance, supports the following `types()`, most of which can be readily used in distributed parallel environments:

C++ code

```
1 {
2 #if !HAVE_MPI && HAVE_SUPERLU
3  "superlu",
4 #endif
5             "bicgstab.amg.ilu0", "bicgstab.amg.ssor", "bicgstab.ilut",
6  "bicgstab.ssor", "bicgstab"
7 #if HAVE_UMFPACK
8                                 , "umfpack"
```

© by the authors, 2017

```
9  #endif
10 }
```

6 The dune-xt-functions module: Local and localizable functions

A correct interpretation of data functions and a correct handling of analytical and discrete functions is an important part of every PDE solver. Given a domain $\Omega \subset \mathbb{R}^d$ and a grid view τ_h of this domain, one is usually not interested in considering a function $f : \Omega \to \mathbb{R}$, but rather its *local function* $f^t := f|_t \circ \Phi^t$ with respect to a grid element $t \in \tau_h$, where $\Phi^t : \hat{t} \to t$ denotes the reference mapping from the respective reference element $\hat{t}$ to the actual element t, as provided by `dune-grid`.For simplicity's sake we only present the scalar case here, but he same concepts apply to vector- and matrix-valued functions, as implemented in `dune-xt-functions`. This allows to compute integrals in terms of quadratures from `dune-geometry` and shape functions from `dune-localfunctions` defined on the reference elements of a grid. In addition, one is interested in "localized derivatives" of f, for instance its *localized gradient* $\nabla_t f_t := \nabla f \circ \Phi^t$. While the localized gradient of a function does not coincide with the gradient of a local function it is an important tool to preserve the structure of an integrand, when transformed to the reference elements for integration. Using the chain rule, we obtain for the latter $\nabla f^t = \nabla(f \circ \Phi^t) = (\nabla f \circ \Phi^t)\nabla\Phi^t$, while the following holds for the former: $\nabla_t f^t = \nabla f \circ \Phi^t = \nabla(f \circ \Phi^t)(\nabla\Phi^t)^{-1} = \nabla f^t (\nabla\Phi^t)^{-1}$.

Consider for instance the following integral arising in the weak formulation of a Laplace operator, which is transformed to the reference element $\hat{t}$, where a, φ and ψ denote scalar functions and $\Lambda_t^2 := |det(\nabla\Phi^{t\perp}\nabla\Phi^t)|$:

$$\int_t (a\nabla\psi) \cdot \nabla\varphi = \int_{\hat{t}} \Lambda_t \Big((a \circ \Phi^t)(\nabla\psi \circ \Phi^t)\Big) \cdot (\nabla\varphi \circ \Phi^t).$$

Using the definition of a local function and a localized gradient, the above is equivalent to

$$\ldots = \int_{\hat{t}} \Lambda_t \, (a^t \nabla_t \psi^t) \cdot \nabla\varphi^t, \tag{2}$$

which nicely preserves the structure of the original integrand.

For a given grid view τ_h, we thus call a function *localizable* with respect to τ_h, if there exists a local function with localized derivatives for each element of the grid view. These functions form the discontinuous space of locally polynomial functions with varying polynomial degree,

$$Q(\tau_h) := \Big\{ q : \Omega \to \mathbb{R} \;\Big|\; \forall t \in \tau_h \; \exists k(t) \in \mathbb{N}, \text{such that } q^t \in \hat{\mathbb{P}}_{k(t)}(t) \Big\},$$

where $\hat{\mathbb{P}}_k(t)$ denotes the set of polynomials $q : \hat{t} \to \mathbb{R}$ of maximal order k. Note that in the context of integration, the polynomial degree of the integrand is limited by the availability of suitable quadratures and always finite in practical computations. Thus, even functions such as $f(x) = \sin(x)$ can in practical computations only be represented by surrogates $f \approx \tilde{f} \in Q(\tau_h)$. Consequently, the space $Q(\tau_h)$ presents a common space for all functions, including analytically given data functions and, most importantly, discrete functions of any discretization framework.

This concept allows for a generic discretization framework such as `dune-gdt`, where all operators, functionals, projections and prolongations are implemented in terms of localizable functions. Thus, a Laplace operator which locally realizes (2) can be used to assemble a system or product matrix, where ψ^t and φ^t are basis functions of a discrete function space, or to compute the norm of any combination of analytical or discrete functions, in which case ψ^t and φ^t model the corresponding local functions. In `dune-xt-functions`, we provide interfaces and implementations to realize this concept of *localizable functions* and *local functions* (not to be confused with shape functions from `dune-localfunctions`) and for the remainder of this section we discuss the realization of these

 © by the authors, 2017

concepts in dune-xt-functions. A similar effort has been undertaken in the dune-functions module [13], independently of our work. Since both efforts share the same mathematical basis it is planned to make dune-xt-functions compatible to dune-functions in future work.

Given a grid element $t \in \tau_h$, we model a collection of (scalar-, vector- or matrix-valued) local functions $\varphi^t : \hat{t} \to \mathbb{R}^{r \times c}$, for $r, c \in \mathbb{N}$ (where $\hat{t}$ denotes the reference element associated with t) by the LocalfunctionSetInterface in dune/xt/functions/interfaces.hh:

C++ code

```
template<class EntityType,
         class DomainFieldType, size_t dimDomain,
         class RangeFieldType,  size_t dimRange, size_t dimRangeCols = 1>
class LocalfunctionSetInterface
{
  // not all methods and types show ...
public:
  virtual const EntityType& entity() const

  virtual size_t size() const = 0;
  virtual size_t order() const = 0;

  virtual void evaluate(const DomainType& xx, std::vector<RangeType>& ret) const = 0;
  virtual void jacobian(const DomainType& xx,
                        std::vector<JacobianRangeType>& ret) const = 0;
};
```

The template parameter EntityType models the type of the grid element $t \in \tau_h$, the parameters DomainFieldType and dimDomain model $\mathbb{R}^d \supset t$ while RangeFieldType, dimRange and dimRangeCols model $\mathbb{R}^{r \times c}$.[38] The resulting DomainType is a FieldVector<DomainFieldType, dimDomaim>, while RangeType and JacobianRangeType are composed of FieldVector and FieldMatrix, depending on the dimensions. Each set of local functions has to report its polynomial order and size. The methods evaluate and jacobian expect vectors of size size() for ret.

Note that we use a combination of static polymorphism, to fix the type of the grid and all dimension at compile time, and dynamic polymorphism, which allows to exchange functions of same grid and dimensions at runtime.

By implementing all operators and functionals in terms of LocalfunctionSetInterface, a discretization framework can realize the above claim of unified handling of analytical and discrete functions. This is for instance the case in dune-gdt, where the local bases of discrete function spaces are realized as implementations of LocalfunctionSetInterface. In addition, we also provide an interface for individual local functions, which can be used by data functions and local functions of discrete functions:

C++ code

```
template <class E, class D, size_t d, class R, size_t r, size_t rC>
class LocalfunctionInterface
  : public LocalfunctionSetInterface<E, D, d, R, r, rC>
{
  // not all methods and types shown ...
public:
  virtual void evaluate(const DomainType& xx, RangeType& ret) const = 0;
  virtual void jacobian(const DomainType& xx, JacobianRangeType& ret) const = 0;

  virtual size_t size() const override final
  {
    return 1;
  }
}
```

[38] As a shorthand, we write <E, D, d, R, r, rC> instead of <EntityType, DomainFieldType, dimDomain, RangeFieldType, dimRange, dimRangeCols>.

Alongside, we also provide the following interface for *localizable functions* which mostly act as containers of local functions:

C++ code

```
template <class E, class D, size_t d, class R, size_t r, size_t rC>
class LocalizableFunctionInterface
{
  // not all methods and types shown ...
public:
  virtual std::string name() const;

      virtual std::unique_ptr< LocalfunctionInterface<E, D, d, R, r, rC>>
  local_function(const EntityType& entity) const = 0;
};
```

Based on this interface we provide visualization and convenience operators, allowing for

C++ code

```
(f - f_h).visualize(grid_view, "difference");
```

where `f` may denote a localizable data function and `f_h` might denote a discrete function, if both are localizable with respect to the same `grid_view`. Expressions such as `f - f_h`, `f + f_h` or `f*f_h` yield localizable functions via generically implemented local functions (if the dimensions allow it).

We also provide numerous implementations of `LocalizableFunctionInterface` in `dune-xt-functions`, most of which can also be created using the `FunctionsFactory`[39], given a `Configuration`:

- `CheckerboardFunction` in `dune/xt/functions/checkerboard.hh` models a piecewise constant function, the values of which are associated with an equidistant regular partition of a domain. Sample configuration for a function $\mathbb{R}^2 \to \mathbb{R}$:

```
lower_left   = [0. 0.]
upper_right  = [1. 1.]
num_elements = [2 2]
values       = [1. 2. 3. 4.]
```

- `ConstantFunction` in `dune/xt/functions/constant.hh` models a constant function. Sample configuration for a function $\mathbb{R}^d \to \mathbb{R}^{2\times 2}$, for any $d \in \mathbb{N}$, mapping to the unit matrix in $\mathbb{R}^2$:

```
value = [1. 0.; 0. 1.]
```

- `ExpressionFunction` in `dune/xt/functions/expression.hh` models continuous functions, given an expression and order at runtime (expressions for gradients can be optionally provided). Sample configuration for $f : \mathbb{R}^2 \to \mathbb{R}^2$ given by $(x, y) \mapsto (x, \sin(y))$:

```
variable   = x
order      = 3
expression = [x[0] sin(x[1])]
gradient.0 = [1 0]
gradient.1 = [0 cos(x[1])]
```

  Note that the user has to provide the approximation order, resulting in f being locally approximated as a third order polynomial on each grid element.

- `GlobalLambdaFunction` in `dune/xt/functions/global.hh` models continuous functions by evaluating a C++ lambda expression. Sample usage for $f : \mathbb{R}^2 \to \mathbb{R}$ given by $(x, y) \mapsto x$:

C++ code

```
GlobalLambdaFunction< E, D, 2, R, 1 > f([](DomainType x){ return x[0]; },
                                        1);  // <- local polynomial order
```

[39] `dune/xt/functions/factory.hh`

 © by the authors, 2017

- `Spe10::Model1Function` in `dune/xt/functions/spe10.hh` models the permeability field of the SPE10 model1 test case, given the appropriate data file[40].

References

[1] Martin Alkämper, Andreas Dedner, Robert Klöfkorn, and Martin Nolte. The dune-alugrid module. *Archive of Numerical Software*, 4(1), 2016.

[2] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The `deal.II` library, version 8.2. *Archive of Numerical Software*, 3, 2015.

[3] 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, 2008.

[4] 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, 2008.

[5] Peter Bastian, Felix Heimann, and Sven Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (dune). *Kybernetika*, 46(2):294–315, 2010.

[6] M. Blatt and P. Bastian. The iterative solver template library. In B. Kagstrom, E. Elmroth, J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 666–675. Springer Berlin Heidelberg, 2007.

[7] M. Blatt and P. Bastian. On the generic parallelisation of iterative solvers for the finite element method. *International Journal of Computational Science and Engineering*, 4(1):56–69, 2008.

[8] M. D. Buhmann. Radial basis functions. In *Acta numerica, 2000*, volume 9 of *Acta Numer.*, pages 1–38. Cambridge Univ. Press, Cambridge, 2000.

[9] A. Burri, A. Dedner, D. Diehl, R. Klöfkorn, and M. Ohlberger. *Advances in High Performance Computing and Computational Sciences: The 1st Kazakh-German Advanced Research Workshop, Almaty, Kazakhstan, September 25 to October 1, 2005*, chapter A general object oriented framework for discretizing non-linear evolution equations, pages 69–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[10] P.G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland Publishing Company, 1978.

[11] J. O. Coplien. Curiously recurring template patterns. *C++ Report*, 7(2):24–27, 1995.

[12] Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90(3):165–196, 2010.

[13] Christian Engwer, Carsten Gräser, Steffen Müthing, and Oliver Sander. The interface for functions in the dune-functions module. *arXiv*, 1512.06136, 2015.

[14] D. Gottlieb and S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1977.

[15] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[40] Available at http://www.spe.org/web/csp/datasets/set01.htm

© by the authors, 2017

[16] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[17] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[18] Dominic Kempf and Timo Koch. System testing in scientific numerical software frameworks using the example of dune. In *Submitted to the Proceedings of the DUNE User Meeting 2015, Sep. 28-29, 2015, Heidelberg, Germany*, 2016.

[19] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[20] A. Logg, K.-A. Mardal, and G. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2012.

[21] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[22] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[23] Scott Meyers. More effective c++: 35 new ways to improve your programs and designs. 1995.

[24] René Milk, Stephan Rave, and Felix Schindler. pymor - generic algorithms and interfaces for model order reduction. *arXiv*, 1506.07094, 2015.

[25] Martin Nolte. *Efficient Numerical Approximation of the Effective Hamiltonian*. PhD thesis, University of Freiburg, 2011.

[26] C. Prud'homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. FEEL++: a computational framework for Galerkin methods and advanced numerical methods. In *CEMRACS'11: Multiscale coupling of complex models in scientific computing*, volume 38 of *ESAIM Proc.*, pages 429–455. EDP Sci., Les Ulis, 2012.

[27] K. Urban. *Wavelet methods for elliptic partial differential equations*. Numerical mathematics and scientific computation. Oxford University Press Oxford, 2009. ISBN 978-0-19-852605-6.

 © by the authors, 2017

Archive of Numerical Software
vol. 5, no. 1, 2017, pages 217–244
DOI: 10.11588/ans.2017.1.28490

The Dune FoamGrid implementation for surface and network grids

Oliver Sander[1], Timo Koch[2], Natalie Schröder[2], and Bernd Flemisch[2]

[1]TU Dresden, Institute for Numerical Mathematics, oliver.sander@tu-dresden.de
[2]University of Stuttgart, Institute for Modelling Hydraulic and Environmental Systems,
{timo.koch, natalie.schroeder, bernd.flemisch}@iws.uni-stuttgart.de

Received: February 29th, 2016; **final revision:** July 9th, 2016; **published:** March 6th, 2017.

Abstract: We present FoamGrid, a new implementation of the Dune grid interface. FoamGrid implements one- and two-dimensional grids in a physical space of arbitrary dimension, which allows for grids for curved domains. Even more, the grids are not expected to have a manifold structure, i.e., more than two elements can share a common facet. This makes FoamGrid the grid data structure of choice for simulating structures such as foams, discrete fracture networks, or network flow problems. FoamGrid implements adaptive non-conforming refinement with element parametrizations. As an additional feature it allows removal and addition of elements in an existing grid, which makes FoamGrid suitable for network growth problems. We show how to use FoamGrid, with particular attention to the extensions of the grid interface needed to handle non-manifold topology and grid growth. Three numerical examples demonstrate the possibilities offered by FoamGrid.

1 Introduction

Various simulation problems are posed on domains that are not open subsets of a Euclidean space. Frequently, such domains are surfaces or curves embedded in a higher-dimensional Euclidean space. Equations on such domains, sometimes called geometric partial differential equations, comprise diffusion and transport on the surface [Dziuk and Elliott, 2007b], flow problems [Nitschke et al., 2012, Reuther and Voigt, 2015], and phase-field equations [T. Witkowski, 2012]. Sometimes, movement of the surface itself is modeled [Dziuk and Elliott, 2007a], and this movement may couple with processes on the surface [Gross and Reusken, 2011].

As an additional difficulty, some boundary value problems are posed on domains Ω that do not even have the structure of a topological manifold. That is, not every point of Ω has a neighborhood that is homeomorphic to an open subset of $\mathbb{R}^d$. Figure 1 illustrates this: While the surface patch on the left is locally homeomorphic to Euclidean space, the one in the middle is a T-junction, and the one on the right is a touching point. Two-dimensional domains with such features appear in applications like the simulation of closed-cell foams [Nammi et al., 2010], or networks of fractures in rock mechanics [McClure and Horne, 2013, McClure et al., 2015]. Of considerable importance are also one-dimensional networks embedded into a two- or three-dimensional Euclidean space.

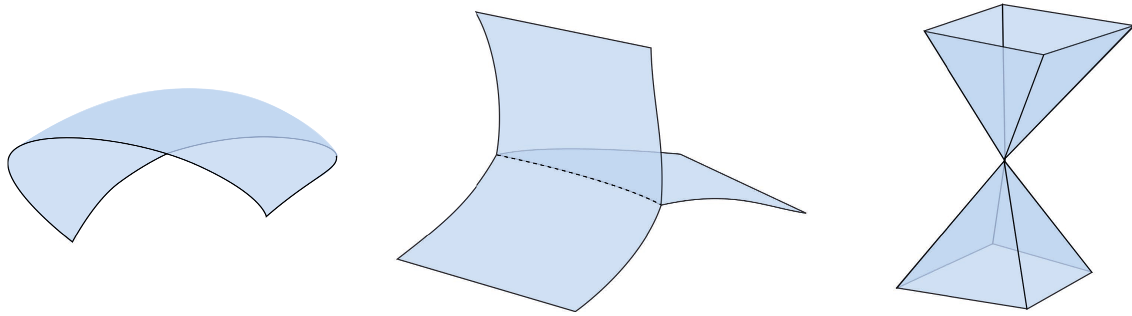


Figure 1: Computational domains might not be topological manifolds. Left: manifold; center: T-junction; right: touching point

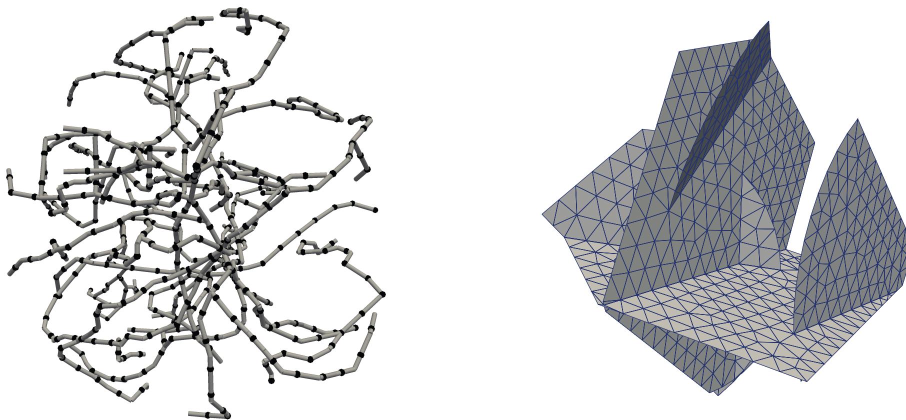


Figure 2: One- and two-dimensional network grids

They appear in models of traffic networks [Garavello and Piccoli, 2006], supply chains [D'Apice et al., 2010], but also in simulations of biological systems like root networks [Dunbabin et al., 2013], neural networks [Lang et al., 2011], or blood vessel networks [Cattaneo and Zunino, 2014a]. An overview over flow problems on networks is given in [Bressan et al., 2014]. Figure 2 shows two example domains for network problems.

Various discretization methods have been proposed for surface and network equations [Dziuk and Elliott, 2013, Olshanskii et al., 2009]. Explicit discretizations, which are the focus of this work, use a grid of the same dimension as the domain. For manifold surface grids it is reasonably simple to generalize grid data structures to such a setting. The main hurdles are admitting that the number of coordinates of a vertex can be different from the effective grid dimension, and making sure that grids are not required to have a boundary. Several standard simulation codes support such surface grids. We mention Alberta [Schmidt and Siebert, 2005] (standalone and as part of Dune), AMDiS [Vey and Voigt, 2007], and FEniCS [Rognes et al., 2013]. Moreover, the GeometryGrid Dune meta grid allows to embed any Dune grid into a Euclidean space of higher dimension.

Grid data structures for non-manifold grids are more challenging. To handle T-junctions, for example, the data structure must cope with the fact that element facets (i.e., edges in a 2d grid) may have more than two neighbors (Figure 1). While this is not very difficult to implement, it requires the introduction of additional data fields and logic, which is not used when the grid happens to be a manifold. Since the latter case is predominant, such additional features mean space and run-time overhead for most users. Therefore, standard grid data structures do not allow

 © by the authors, 2017

for non-manifold topologies. Numerical examples of processes on non-manifold topologies in the literature are typically done using ad hoc implementations of the necessary grid data structures.

Unfortunately, using ad hoc implementations wastes a lot of human resources. While the domain may be non-standard, many of the equations on network grids differ little from their Euclidean counterparts. However, existing code like finite element assemblers for diffusion and transport processes cannot be reused on ad hoc grid data structures. They require detailed knowledge of the grid, and it is a challenge to port an existing assembler to a new grid data structure. Also, since the data structure is ad hoc, it is difficult to reuse it in other contexts. In particular, it is difficult to share it with other groups working in the same field.

The Dune software system (see `www.dune-project.org` and [Bastian et al., 2008b,a]) has found an elegant solution for both problems. Dune is a set of open-source C++ libraries dedicated to various aspects of finite element and finite volume methods. Its grid component, implemented in the `dune-grid` module, specifies an abstract interface for computational grids. The specification mandates what a data structure should be able to do to qualify as a Dune grid, and how this functionality should be accessible from C++ code. Examples of such functionality include being able to iterate over the elements and vertices, and getting the maps from the reference element to the grid elements. Those parts of a numerical simulation code that use the grid, such as the matrix assemblers or error estimators, are written to use only the abstract grid interface. Grid data structures implementing this interface are then completely decoupled from the algorithms that use them.

This decoupling has various interesting consequences. Using the Dune grid interface, it is easy to swap a given grid implementation for another one. Indeed, in typical Dune applications the C++ grid type is set once in the code, and handed around as a template parameter. Changing the initial **`typedef`** and recompiling the code is usually sufficient to switch to an alternative grid data structure. This possibility to easily switch between grid implementations allows to provide tailor-made grid data structures for special simulation needs. For example, `dune-grid` itself provides YaspGrid, the implementation of a structured grid with very little run-time and space overhead. In contrast, UGGrid implements a very flexible unstructured grid with non-conforming and red–green refinement.

In this paper we present FoamGrid, a new implementation of the Dune grid interface that is dedicated to surface and network grids. Its main features are:

- FoamGrid implements one- and two-dimensional simplex grids embedded in a physical Euclidean space of arbitrary dimension w. Hence, it targets geometric and surface PDEs.
- The grids do not need to have the structure of a topological manifold. Network configurations like the ones in Figures 1 and 2 are supported.
- A FoamGrid can be adaptively refined. In the standard setup, refining a triangle results in four coplanar triangles. Additionally, FoamGrid elements can be parameterized, i.e., they can be given a map that describes a nonlinear embedding of the element into $\mathbb{R}^w$. As the element gets refined more and more, its shape approaches the one described by the parameterization (Figure 4).
- Finally, the domain of a FoamGrid can grow and shrink at run-time. Elements can be added and removed even when there is no coarser "father" element, without invalidating the grid data structure. Data can be transferred during this process. This allows to elegantly simulate network growth and remodeling processes.

FoamGrid is available as a Dune module `dune-foamgrid` and is installed just like any other Dune module. `dune-foamgrid` is free software, available under the either the LGPLv3+, or

© by the authors, 2017

the GPLv2 with a linking exception clause. The git repository is publicly accessible at https://gitlab.dune-project.org/extensions/dune-foamgrid.git and contributions are welcome.

Using Dune and FoamGrid for simulations of network and surface PDE problems has a number of important advantages. First of all, you do not have to implement the grid data structure yourself. While not overly difficult, quite a bit of thought has gone into FoamGrid, which would need work to be replicated. Then, since FoamGrid implements the Dune grid interface, large amounts of existing application and infrastructure code can be used directly with FoamGrid. This includes things like finite element spaces and assemblers, error estimators, and grid file readers and writers. This advantage is demonstrated in particular by the numerical example in Section 4.1, which required no additional coding at all to extend an existing planar code to a network setting.

As a further advantage, once a user starts employing FoamGrid and the Dune grid interface, he immediately has the power of all other Dune grid implementations at his disposal. All of these are easily usable together with FoamGrid. Hence, e.g., in simulations that couple network grids with background grids, several implementations of such background grids are readily available. (Authors's remark: the Dune extension module `dune-grid-glue` offers a convenient way to do the coupling – www.dune-project.org/modules/dune-grid-glue). Since the Dune grid interface is a well-established standard, it is easy to learn how to use FoamGrid. If a user already knows Dune, there is little additional knowledge needed. Since FoamGrid is open source, sharing code based upon it is particularly easy.

While FoamGrid has many interesting features, there is also a number of things it does not currently support. For example, elements can only be simplices, and they must be one- or two-dimensional. (The authors could not think of a use case for a higher-dimensional network grid. Write us if you know one.) Adaptive grid refinement is currently non-conforming, which leads to hanging nodes, and, possibly, to holes in the surface (Figure 5). Finally, the current FoamGrid implementation is purely sequential, and FoamGrid objects cannot be distributed across several processes. However, the development of FoamGrid is ongoing, and these features may appear in later releases.

The present article is structured as follows. Chapter 2 briefly explains how to use FoamGrid. Everything mentioned there is codified in the Dune grid interface specification, and hence you can also read this chapter as an introduction to the use of the Dune grid interface. Chapter 3 then explains the special features that FoamGrid offers. In particular, these are support for T-junctions, adaptive refinement with element parameterizations, and the ability to "grow". Finally, in Chapter 4 we give three numerical examples showcasing the different features of FoamGrid. The first shows unsaturated flow through a two-dimensional fracture network using finite elements. The second example shows h-adaptive, locally mass-conservative transport of a therapeutic agent in a microvascular network using finite volumes. The third one models the growth of plant root networks.

2 FoamGrid and the Dune grid interface

In this chapter, we start by describing the programmer interface of FoamGrid. FoamGrid implements the Dune grid interface, hence in many central aspects, it can be used just like any other Dune grid. This chapter focuses on these aspects. You can therefore also read it as a brief review of the Dune grid interface. For more details, you may want to consult the Dune online documentation and [Bastian et al., 2008b,a].

The central class of the FoamGrid grid implementation is

C++ code

```
1 template <int dim, int dimworld>
2 class FoamGrid;
```

 © by the authors, 2017

available from the header dune/foamgrid/foamgrid.hh. This class implements a hierarchical grid as defined in [Bastian et al., 2008b, Def. 13], i.e., a coarse (or *macro*) grid, and element refinement trees rooted in each of its elements. The first template parameter dim is the grid dimension d, which must be either 1 or 2. The second template parameter dimworld is the dimension w of the Euclidean embedding space. It must be equal to or greater than the grid dimension, but can otherwise be arbitrary. For the rest of this article we use dim and dimworld in code examples, and d and w in text to denote the dimensions of the grid and the physical space, respectively.

To construct FoamGrid objects, the FoamGrid class implements the entire Dune grid interface for the setup of unstructured grids. In particular, the class GridFactory is implemented for FoamGrid. Thus, all file-reading methods based on this interface are available. For example, files in the gmsh format [Geuzaine and Remacle, 2015] can be read by using the line

C++ code

```
std::shared_ptr<FoamGrid<2,3> grid( GmshReader<FoamGrid<2,3>>::read("filename.msh") );
```

This will read the file named "filename.msh", and set up a new FoamGrid<2,3> object with it. The new grid object is returned in a shared pointer called grid. Note that vertex coordinates in gmsh files always have three components, so reading a gmsh file into a FoamGrid<1,2> object will discard the third entry. As a special feature, gmsh files can contain elements with polynomial geometries of order up to five. While FoamGrid element geometries are always affine, FoamGrid can use the higher-order geometries during mesh refinement (Section 3.3).

Writing FoamGrid objects to disk is equally straightforward. All Dune file writing codes rely on the grid interface only, and can therefore be used with FoamGrid. For example, writing the object pointed to by the grid shared pointer into a VTK file called my_filename.vtu can be achieved by including the header dune/grid/io/file/vtk.hh from the dune-grid module, and writing

C++ code

```
typedef FoamGrid<2,3> GridType;
VTKWriter<GridType::LeafGridView> vtkWriter(grid->leafGridView());
vtkWriter.write("my_filename");
```

The resulting file can be visualized, e.g., with the ParaView software.

2.1 Elements and geometries

In Dune, finite element assembly typically takes place on the leaf elements of the refinement trees. These elements form the *leaf grid view*, which encapsulates the notion of a textbook non-hierarchical finite element grid. From a FoamGrid, the leaf grid view can be obtained in the usual way, i.e.,

C++ code

```
auto foamGridLeafView = grid->leafGridView();
```

which already has been used in the VTK example above.

Access to grid elements and vertices is provided by the grid view. Elements can be iterated over using begin/end-iterators

C++ code

```
for (auto it = foamGridLeafView.begin<0>();
     it != foamGridLeafView.end<0>();
     ++it)
{
  // do something with the element in '*it'
}
```

© by the authors, 2017

where the number `0` specifies that the loop is to be over the grid elements (it is the codimension of the grid elements with respect to the grid). Using the C++11 range-based-**for** syntax, the same loop can be written more concisely

C++ code
```
for (const auto& element : elements(foamGridLeafView))
{
  // do something with the element in 'element'
}
```

Similarly, a loop over all vertices of the grid is written as

C++ code
```
for (auto it = foamGridLeafView.begin<dim>();
     it != foamGridLeafView.end<dim>();
     ++it)
{
  // do something with the vertex in '*it'
}
```

In this code, the number `dim` (i.e., the grid dimension), specifies that the loop is to be over the grid vertices, because vertices are zero-dimensional and hence have codimension `dim` in a `dim`-dimensional grid. Alternatively, one can write

C++ code
```
for (const auto& vertex : vertices(foamGridLeafView))
{
  // do something with the vertex in 'vertex'
}
```

The objects `vertex` and `element` (or `*it` in the iterator loops) are instances of what in DUNE terminology are called *entities*. Entities are implemented by the `Entity` interface class in `dune-grid`. They provide topological information about the grid elements and vertices, like links to the corners vertices of an element, and to the father and descendant elements in the refinement tree. In all these aspects, `FoamGrid` objects behave just like any other DUNE grids. Furthermore, each element entity provides a `Geometry` object, which represents the affine map $F : T_{\text{ref}} \to T \subset \mathbb{R}^w$ from the reference element T_{ref} to the grid element T. This map provides the geometrical information needed to assemble finite element and finite volume systems, like evaluation of F and its inverse F^{-1}, the inverse transposed Jacobian matrix ∇F^{-T}, and the functional determinant $J := \sqrt{\det \nabla F^T \nabla F}$. Note that since for surface grids the world dimension w is larger than the dimension of the reference element T_{ref}, the image of T_{ref} under F is a set of measure zero in $\mathbb{R}^w$. In finite-precision arithmetic the argument of the method implementing F^{-1} will typically not be in the domain of F^{-1}. The `Geometry` implementation of FOAMGRID therefore extends F^{-1} to the entire space $\mathbb{R}^w$. Given a point $x \in \mathbb{R}^w$ not necessarily in T, `FoamGrid` computes a point ξ in the plane spanned by T_{ref} such that $|F(\xi) - x|$ is minimized. The affine function F is tacitly extended from T_{ref} to its entire affine hull here.

To compute element fluxes, elements in a DUNE grid provide a set of so-called *intersections*, which relate elements to their neighbors. This mechanism is very flexible, and in particular handles general non-conforming situations very well. Nevertheless, using grids for network domains stretches the bounds of the current intersection concept, and some generalization is needed. We have therefore dedicated a separate chapter to FOAMGRID intersections (Chapter 3.1).

2.2 Attaching data to grids

Data is attached to FOAMGRID objects in the same way as to other DUNE grids. Each grid view object can provide a corresponding `IndexSet` object

 © by the authors, 2017

C++ code

```
const auto& indexSet = foamGridLeafView.indexSet();
```

This index set provides an integer number for each entity (i.e., vertex, edge, or element) of the grid view. For each dimension and reference element type, these numbers are consecutive and start at zero. They can hence be used to address random-access containers holding the simulation data. This approach is convenient, flexible, and efficient.

Data stored in arrays is lost if the grid changes, either by refinement (Section 2.3) or by grid growth (Section 3.2). To preserve data across grid modifications, FoamGrid, just like any other Dune grid, additionally provides a set of persistent numbers. These are obtained by an `IdSet` object

C++ code

```
const auto& idSet = grid->localIdSet();
```

(remember that in our initial example the variable `grid` was a shared pointer to a `FoamGrid`). Persistent numbers are neither consecutive nor restricted to start at zero, but they can be used to access search trees or hash maps. Before modifying the grid, all simulation data must be copied into such data structures, and copied back to arrays after the modification is completed. While such copying is costly, its run-time is usually negligible compared to the cost of the actual grid modification.

2.3 Adaptive refinement

FoamGrid supports red refinement (non-conforming refinement) of simplices, where each triangle is split into four congruent smaller triangles. If a two-dimensional FoamGrid is refined locally, then hanging nodes appear in the grid. Depending on the discretization used, this may or may not be a problem. True red–green refinement, which avoids the hanging nodes, may appear in later versions of FoamGrid.

Adaptive grid refinement in FoamGrid is controlled via the standard Dune grid interface. In a first step the method

C++ code

```
bool mark (int refCount, const Codim<0>::Entity& element);
```

is used to mark an element `element` for refinement (`refCount` > 0) or coarsening (`refCount` < 0). The method returns `true` if the element was successfully marked. The mark of an element `element` can be obtained with the method

C++ code

```
int getMark (const Codim<0>::Entity& element) const;
```

which returns `1` for elements marked for refinement, `-1` for elements marked for coarsening, and `0` for unmarked elements.

The grid is then modified in a second step with the methods

C++ code

```
bool coarsen = grid.preAdapt(); // true if at least one element
                                // will be coarsened
bool refined = grid.adapt();    // true if at least one element
                                // was refined
grid.postAdapt();
```

Between `preAdapt()` and `adapt()` it is possible to check the following flag

© by the authors, 2017

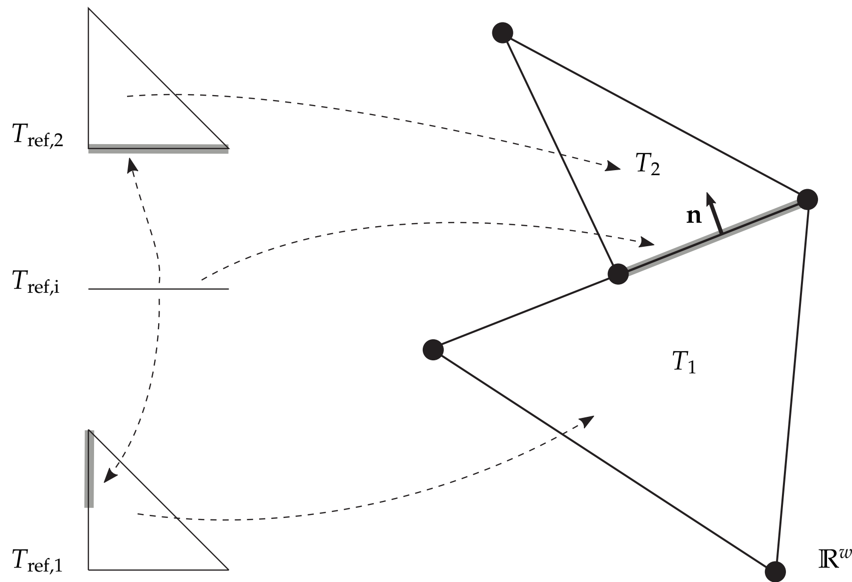


Figure 3: Intersection between two elements T_1 and T_2. From its reference element $T_{\text{ref,i}}$, there are two maps to $T_{\text{ref},1}$ and $T_{\text{ref},2}$ (the reference elements of T_1 and T_2), respectively, and one to $\mathbb{R}^w$. These maps describe the shape of the intersection in T_1, T_2, and $\mathbb{R}^w$, respectively.

C++ code

```
1 bool mightVanish = element.mightVanish(); // true if the element might
2                                           // vanish due to coarsening
3                                           // by grid.adapt()
```

Similarly, between `adapt()` and `postAdapt()` one can check the flag

C++ code

```
1 bool isNew = element.isNew(); // true if element was created by last
2                               // call to adapt()
```

Both are useful to manage the transfer of data associated with the grid. As FoamGrid itself does not store any associated data, the data transfer from the old grid to the adapted grid is managed by the user. An example featuring adaptive refinement and coarsening is presented in Section 4.2.

3 Distinctive and novel features of the FoamGrid implementation

The previous chapter has described those aspects where FoamGrid behaves just like any other Dune grid. However, FoamGrid also has a few features that set it apart from most other Dune grids. The present chapter is dedicated to those features.

3.1 Handling intersections in a non-manifold grid

The defining feature of FoamGrid is its capability to handle network grids, i.e., grids with a non-manifold topology (Figures 1 and 2). In particular, more than two elements can meet in a common vertex in a one-dimensional grid, and more than two triangles can meet in a common edge in a two-dimensional grid.

Information about how a given element relates to its neighbors is essential for finite volume and DG methods, or more generally all methods involving element boundary fluxes. For this, the Dune grid interface provides the notion of *intersections*. In Dune terminology, an *intersection* is the

 © by the authors, 2017

set-theoretic intersection between the closures of two neighboring elements. Only intersections that have positive $(d-1)$–dimensional measure qualify as intersections in the Dune sense, and those are equipped with a coordinate system, i.e., a map from a $(d-1)$–dimensional reference element $T_{\text{ref,i}}$ to the intersection (Figure 3). Note that if the grid is conforming, then intersections will be geometrically the same as shared element faces. However, in general non-conforming grids, intersections are only subsets of element faces.

Intersections provide all information needed to compute element boundary fluxes. In particular, for each point on an intersection, you can get the vector $\mathbf{n}$ that is tangent to the element at that point, and normal to the element boundary. Also, you get the embeddings of the intersection into the two elements, in form of maps from the intersection reference element $T_{\text{ref,i}}$ to the element reference elements $T_{\text{ref,tr}}$ (Figure 3). Those maps pick up the same ideas used to model element geometries in Section 2.1. They are called *geometry-in-inside* and *geometry-in-outside*, respectively.

In C++ code, *intersections* are objects of type `Intersection`. For any given element (which is then called the *inside* element), all its intersections with neighboring elements can be accessed by traversing them with a dedicated iterator. Using range-based **for** syntax, a loop over all intersections of the element called `element` is written as

C++ code

```
1 for (const auto& intersection : intersections(foamGridLeafView,element))
2 {
3   // do something with 'intersection'
4 }
```

If you iterate over all intersections of all elements, then you will encounter each intersection twice, but once with its role reversed. An exception are the intersections of elements with the domain boundary, which also count as Dune intersections, but which are only accessible from one element. See the `dune-grid` class documentation for the details on the user interface of the `Intersection` class.

Unfortunately, the Dune intersection mechanism, as flexible as it is, is not flexible enough for network grids. The original idea was that while elements may have more than one neighbor across a given facet, there is (even in non-conforming situations) at most one neighbor *at any given point* on that facet. This reflects the assumption that computational domains are expected to be topological manifolds. At the time of writing, this assumption is still reflected in the grid interface. In particular, the method

C++ code

```
1 bool neighbor() const;
```

(a public member of the `Intersection` class), informs whether there is a neighbor across a given intersection. However, this information is insufficient in network grids, where even in a conforming grid there may be an arbitrary number of neighbors meeting at a common intersection. Finite volume and DG methods need access to this group of neighbors, to know how to distribute the flux across this intersection.

For this reason the intersection concept and programmer interface is being revisited, and is likely to undergo changes in the future to better support network grids. Changes to the Dune interface can only be made in a democratic process, and it is therefore unclear at what point in time such a change will happen.

Two different approaches have been proposed to the Dune grid development community. We describe them both briefly here. The full proposal text is available at `www.dune-project.org/modules/dune-foamgrid`.

The first approach tries to be as minimally invasive as possible. In particular, it retains the notion of an intersection as an object that relates *two* elements. The extension consists of two

© by the authors, 2017

semantic rules, and a change to the `neighbor` method. The first of the semantic rules makes sure that the "number of neighbors" across a given intersection is well-defined. Remember that the *geometry-in-inside* is, roughly speaking, the intersection interpreted as a subset of the element T_1.

Rule 1 *For any two intersections of a given element T_1, the geometries-in-inside are either disjoint or identical.*

With the number of neighbors properly defined, the `neighbor` method is generalized to return this number instead of a yes/no answer:

C++ code

```
1 std::size_t neighbor() const;
```

The number of intersections with identical geometries-in-inside will be equal to the value returned by `neighbor()` for each of those intersections. Note that this change is fully backward-compatible, as intersections in a non-network grid will return either `1` or `0` here, which casts to the values `true` or `false` as used previously.

The second semantic rule provides a way to find all intersections that share a common geometry-in-inside. No additional interface method is added for this. Rather, it is guaranteed that all such intersections appear consecutively when traversing the intersections with the intersection iterator.

Rule 2 *If more than one neighbor is reachable over a given geometry-in-inside, then all intersections for this geometry-in-inside shall be traversed consecutively by the intersection iterator.*

With this rule, groups of neighbors can be identified and used in flux computations.

The second approach is more radical, because it changes the idea of an intersection itself. Intersections cease to be objects that relate *pairs* of elements. Rather, they now become objects that relate *groups* of elements. As a consequence, each intersection still has only one geometry-in-inside. However, for each intersection there is now more than one outside element, each with corresponding geometry-in-outside and index-in-outside.

To access this information, more interface methods need to be changed. First of all, the `neighbor` method needs to be changed as proposed above. Secondly the methods

C++ code

```
1 Entity outside () const;
2 LocalGeometry geometryInOutside () const;
3 int indexInOutside () const;
```

of the `Intersection` interface class need to be replaced by

C++ code

```
1 Entity outside (std::size_t i=0) const;
2 LocalGeometry geometryInOutside (std::size_t i=0) const;
3 int indexInOutside (std::size_t i=0) const;
```

respectively. Rather than returning the unique outside element or its geometry or index, the methods must now return the corresponding quantity for the `i`-th outside element.

These changes are again fully backward-compatible. In grids without multiple intersections, at most the 0-th outside element will be available. The default parameter ensures that this intersection will be returned when the method is called without argument.

As an advantage, this proposal retains the rule that the geometries-in-inside must form a disjoint partition of the element boundary (modulo zero-sets). Also, it is easier to attach data to such

 © by the authors, 2017

intersections, which is a feature that has been requested various times in the past. On the downside, to iterate over all neighbors of an element, two nested loops are needed, instead of only one. This will make some code a bit longer, and more difficult to read.

Both proposals are currently under discussion. However, even with the current status quo, applications involving fluxes can be written for network domains. One-dimensional domains are straightforward as there the intersections are a fortiori conforming (see Sections 4.2 and 4.3). Two-dimensional networks need more trickery, but can also be made to work. Once either of the proposed interface extensions has been officially accepted, implementations of such methods will be much simpler.

3.2 Grid growth

A certain number of network problems is posed on domains that grow and/or shrink in the course of the simulation. Examples are fracture growth processes and simulations of bone trabeculae remodeling. To support such simulations, FoamGrid objects are allowed to grow and shrink, i.e., elements can be added and removed from the grid at runtime. Finite element and finite volume data is kept during such grid changes, using an approach much like the one used for grid adaptivity. Of all Dune grids, FoamGrid is currently the only one to support this feature. The programmer interface combines ideas from the `GridFactory` class to insert new vertices and elements, with the adaptivity interface to allow to keep data across steps of grid growth.

Growth and shrinkage of grids is a two-step process. First, new elements and vertices are handed to the `FoamGrid` object. These are not inserted directly; rather, they are queued for eventual insertion. In addition, individual elements can be marked for removal. Once all desired elements and removal marks are known to the grid, the actual grid modification takes place in a second step.

Queuing elements for insertion and removal is controlled by three methods. The first,

C++ code

```
1 unsigned int insertVertex(const FieldVector<double,dimworld>& x);
```

queues a new vertex with coordinates `x` for insertion. The return value of the method is an index that can be used to refer to this vertex when inserting new elements. The index remains fixed until all queued elements are actually inserted in the grid (by the `grow` method), but may change during the execution of that method. Inserting elements is done using

C++ code

```
1 void insertElement(const GeometryType& type,
2                    const std::vector<unsigned int>& vertices);
```

which mimics the corresponding method from the `GridFactory` class. The argument `type` has to be a simplex type, because (currently) FoamGrid supports only simplex elements. The array `vertices` must contain the indices of the vertices of the new element to be inserted. These can be either indices of existing vertices, or new indices obtained as the return values of the `insertVertex` method.

Analogously a new element with a parametrization can be inserted by calling

C++ code

```
1 void insertElement(const GeometryType& type,
2                    const std::vector<unsigned int>& vertices,
3                    const std::shared_ptr<VirtualFunction<
4                                            FieldVector<ctype,dim>,
5                                            FieldVector<ctype,dimworld>
6                                       > >& elementParametrization);
```

Finally, the method

C++ code
```
void removeElement(const Codim<0>::Entity& element);
```

marks the given element for removal.

Once all desired elements are queued for insertion or removal, the actual grid modification takes place in a second step. The grid is modified using the method

C++ code
```
bool elementsInserted = grid->grow(); // true if at least one element was inserted
```

While element removal is guaranteed, queuing elements does not assure that the element will be inserted. New elements are restricted by the fact that DUNE grids are hierarchic objects. The vertices given by the user to form an element are always leaf vertices but may be contained in different hierarchic levels. However, elements can only be constituted by vertices of the same level. Therefore, new elements in FOAMGRID are always inserted on the lowest possible level substituting the given vertex by its hierarchic descendants or ancestors. Note that it is not generally guaranteed that relatives of the given vertices on the same level can be found. In that case, the element will not be inserted. The method `grow` will return `true` if it was possible to insert at least one element.

After the call to `grow`, it is possible to check whether a given element has been created by the last call to the `grow` method:

C++ code
```
bool isNew = element.isNew();    // true if element was created by last growth step
```

which is a method of the interface class `Entity<0>`, i.e. elements. Observe that this is the same method that returns whether an element has been created by grid refinement. Hence its semantics depends on whether it is queried after a call to `grow` or after a call to `adapt`. Using this method is helpful, e.g., when setting initial values and/or boundary conditions for newly created elements.

The growth is completed with the call

C++ code
```
grid->postGrow();
```

which removes all `isNew` markers.

Summing up, growing or shrinking the grid while keeping grid data consists of the following steps. Note the relationship to grid adaptivity with data transfer.

1. Mark elements for removal; queue new vertices and elements for insertion.
2. Transfer all simulation data attached to the grid into an associative container indexed by the entity ids described in Section 2.2.
3. Call `grow()`.
4. Resize data array; copy data from the associative container into the array.
5. Set initial data at newly created elements and vertices and boundary conditions at newly formed boundaries. Note that element removal always creates new boundaries.
6. Finalize by calling `postGrow()`.

 © by the authors, 2017

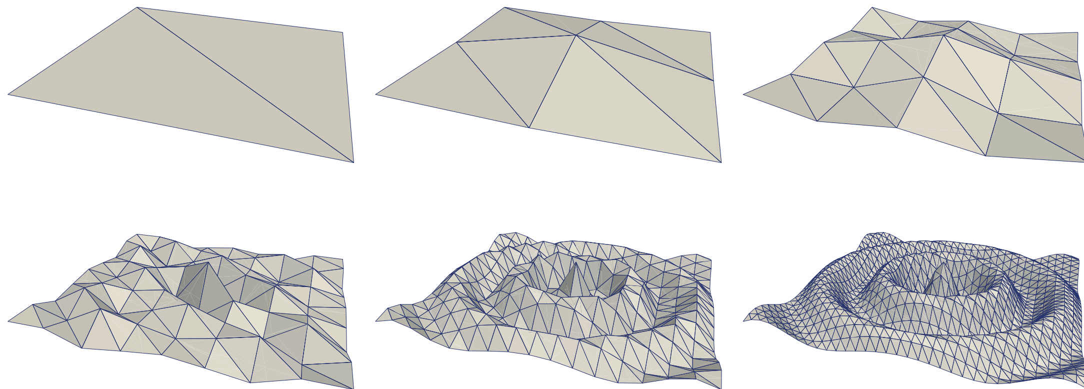


Figure 4: Grid refinement with element parametrizations

While grid growth itself is straightforward, it is difficult to use in combination with adaptive refinement. Grid refinement in Dune leads to hierarchical grids, which are forests of refinement trees. Not every element of such a forest can be added or removed without violating certain consistency conditions. In one-dimensional grids, there are relatively few problems, and grid growth and refinement can be used together to good effect. For two-dimensional grids we have tried to be as general as possible, but there are limits.

There are obstacles both to the removal of elements and to the insertion of new ones, if grid refinement is involved. First of all, only leaf grid elements can be removed. There is no conceptual problem with element removal if the element has no father. If it does have a father, however, removing only a subset of its 2^d sons is a violation of the Dune grid interface specification, which mandates that the sons of an element must (logically) cover their father [Bastian et al., 2008b, Def. 11.1]. Deliberately allowing this violation nevertheless appears to be the only viable solution, as all other possibilities amount to only allowing the removal of large groups of elements, which is rarely desired.

Inserting new elements into a refinement forest of elements is even more problematic, because the new element needs to be assigned a level number in the hierarchy. Ideally, this level would always be zero, because if the element had a larger level number it would be expected to have a father. FoamGrid does violate this assumption if necessary, which does not lead to problems in practice, unless multigrid-type algorithms are used. On the other hand, the element level must be the same as the levels of all of its vertices. If the vertices are new, their levels can be freely chosen. However, grid growth almost always involves vertices that already exist in the grid. When trying to insert elements with vertices having different level numbers, FoamGrid currently tries to replace existing vertices by their father vertices, to obtain a set of $d+1$ vertices on the same level (which then determines the element level).

3.3 Element parametrizations

In a standard non-conforming red refinement algorithm, new vertices are placed at the edge midpoints of refined elements. That way, while the grid gets finer and finer, the geometry of the grid remains identical to the geometry of the coarsest grid. To also allow improvements to the geometry approximation, FoamGrid can use element parametrizations. Each coarse grid element T with reference element T_{ref} of a FoamGrid can be given a map

$$\varphi_T : T_{\text{ref}} \to \mathbb{R}^w,$$

which describes an embedding of T into physical space $\mathbb{R}^w$. This does not influence the grid itself — elements of a FoamGrid are always affine. However, when refining the grid, new elements are not inserted at edge midpoints. Rather, for each new vertex which would appear at an edge midpoint in standard refinement, the corresponding local position in its coarsest ancestor element T is determined using the refinement tree. This position is then used as an argument for φ_T, and the result is used as the position of the new vertex. That way, the grid approaches the shape described by the parametrization functions φ_T more and more as the grid gets refined (Figure 4).

In `dune-grid`, element parametrizations are implemented as small C++ objects. These must inherit from the abstract base class

C++ code

```
1  VirtualFunction<FieldVector<double,dim>,
2                  FieldVector<double,dimworld> >;
```

declared in `dune/common/function.hh`.

One object of this type needs to be created for each element of the coarse grid. The base class has a single pure virtual method

C++ code

```
1  virtual void evaluate(const FieldVector<double,dim>& x,
2                        FieldVector<double,dimworld>& y) const = 0;
```

which implements the evaluation of φ_T. The first argument `x` is a position in local coordinates of the corresponding coarse grid triangle. The result $\varphi_T(x)$ is returned in the second argument `y`.

Element parametrizations are handed to the `GridFactory` during grid construction. Normally, grid elements are entered using the method

C++ code

```
1  void insertElement(const GeometryType& type,
2                     const std::vector<unsigned int>& vertices);
```

of the `GridFactory` class. For parametrized elements, there is the alternative method

C++ code

```
1  void insertElement(const GeometryType& type,
2                     const std::vector<unsigned int>& vertices,
3                     const std::shared_ptr<VirtualFunction<
4                                          FieldVector<ctype,dim>,
5                                          FieldVector<ctype,dimworld>
6                                       > >& elementParametrization);
```

This inserts the element with vertex numbers given in the array `vertices` and an element parametrization given by the object `elementParametrization` into the grid. The `GmshReader` uses this method for some of the higher-order elements that can appear in GMSH grid files.

To see the effect of element parametrizations, Figure 4 shows an example where the coarsest grid consists of only two triangles covering the domain $\Omega = [-1, 1]^2 \times \{0\}$. As a parametrization we use the global function

$$\varphi : \mathbb{R}^2 \to \mathbb{R}^3, \qquad \varphi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ 0.2 \cdot \exp(-|x|) \cos(4.5\pi|x|) \end{pmatrix}, \tag{1}$$

and each element parametrization φ_T first maps its local coordinates to $\mathbb{R}^2$, and then applies the global function φ. Hence, upon refinement, the grid will approach the graph of the function (1). The code for this example is provided in the `dune-foamgrid` module itself, in the file `dune-foamgrid/examples/parametrized-refinement.cc`.

 © by the authors, 2017

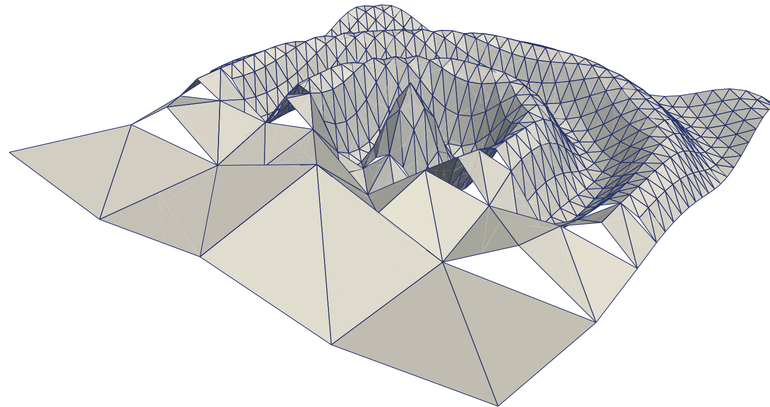


Figure 5: Adaptive refinement with element parameterizations leads to non-conforming geometries

There is one pitfall when using element parametrizations together with non-conforming adaptive refinement for two-dimensional grids. If a hanging node appears in the course of refinement, the position of this node is determined by the parametrization functions of the corresponding element, which is typically *not* the midpoint of the adjacent longer edge. As a consequence, the grid will have holes wherever different refinement levels meet (Figure 5). While this may seem surprising at first sight, it is nevertheless a logical consequence of the hanging nodes refinement. The continuous domain is approximated by discontinuous grids, in direct correspondence to how discontinuous FE functions may be used to approximate continuous PDE solutions.

The non-conforming geometry approximation shows another shortcoming of the current Dune intersection concept. There, intersections are defined as set-theoretic intersections of (closures of) neighboring elements. However, in the geometrically non-conforming situation, neighboring elements do not actually intersect, and one can only speak of logical intersections. The practical consequence of this is that intersections do not have a single well-defined shape in $\mathbb{R}^w$ anymore. Rather, there are now *two* of them. This possibility to have two global element shapes is not currently reflected by the Dune grid interface. In the current implementation of FoamGrid, the method `Intersection::Geometry` will always return the global shape of the intersection as seen from the inside element.

3.4 Moving grids

Many interesting geometric PDE problems are posed on surfaces that change their shape over time. Discretization of such PDEs can require a surface grid that can move and deform during a simulation (see, e.g., [Dziuk and Elliott, 2007a]). FoamGrid caters to such applications by providing a method that allows to reset the position of any grid vertex at any time. This method is a public member of the `FoamGrid` class and has the signature

C++ code

```
1 void setPosition(const Traits::Codim<dimgrid>::Entity& vertex,
2                  const FieldVector<ctype, dimworld>& pos);
```

It will reset the position of the vertex given in `vertex` to the position given in `pos`.

While `setPosition` can be called for grid vertices on any level, it is really only advisable to call it for vertices from the leaf grid view. Recall that in a globally or locally refined Dune grid, copies of the same vertex exist on different refinement levels. The `setPosition` method will set the

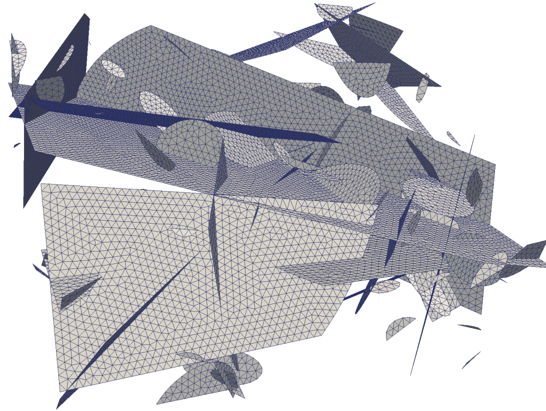


Figure 6: Grid for a discrete fracture network, courtesy of Patrick Laug. The network and grid were generated using the software described in [Borouchaki et al., 2000].

position of any ancestor and descendant vertices of the argument vertex to the position given in the `pos` argument. That way, the grid hierarchy remains consistent, and the grid will not "move back" to its original shape after eventual coarsening. However, this only works if `setPosition` is called for the leaf vertices.

4 Numerical examples

We close the article with three numerical examples. These show how seemingly challenging algorithms can be implemented with ease using the FoamGrid grid manager.

4.1 Unsaturated Darcy flow in a discrete fracture network

For our first example we simulate unsaturated Darcy flow through a network of two-dimensional fractures embedded into $\mathbb{R}^3$. We only consider the flow in the network itself, but FoamGrid can be easily coupled to higher-dimensional background grids using the `dune-grid-glue` module [Bastian et al., 2010, Engwer and Müthing, 2016].

Let the computational domain Ω be the union of a finite number of closed, bounded hypersurfaces in $\mathbb{R}^3$. We use the Richards equation to model the flow in Ω [Bear, 1988]. That is, we suppose that the state of the system in a time interval $[0, T]$ can be described by a scalar pressure head

$$p : \Omega \times [0, T] \to \mathbb{R}.$$

From the pressure head, the volumetric water content θ and relative permeability kr can be computed using the Brooks–Corey and Burdine parameter functions

$$\theta(p) = \begin{cases} \theta_m + (\theta_M - \theta_m)\left(\frac{p}{p_b}\right)^{-\lambda} & \text{for } p \leq p_b \\ \theta_M & \text{for } p \geq p_b, \end{cases} \qquad \mathrm{kr}(\theta) = \left(\frac{\theta - \theta_m}{\theta_M - \theta_m}\right)^{3+\frac{2}{\lambda}},$$

where θ_m, θ_M, p_b, and λ are scalar parameters. The water content θ satisfies the Richards equation

$$\frac{\partial}{\partial t}\theta(p) + \operatorname{div} \mathbf{v}(x, p) = 0, \qquad \mathbf{v}(x, p) = -K_f(x)\,\mathrm{kr}(\theta(p))\nabla(p + z). \tag{2}$$

For simplicity we suppose that the flow is purely driven by the boundary conditions, and omit the gravity term z.

 © by the authors, 2017

Equation (2) is a quasilinear equation in the pressure head p. In [Alt and Luckhaus, 1983] (see also [Berninger et al., 2011]) it was shown how the Kirchhoff transformation can be used to transform it to a semilinear equation for a generalized pressure

$$u : \Omega \times [0, T] \to \mathbb{R}, \qquad u(x,t) = u(p(x,t)) := \int_0^p \mathrm{kr}(\theta(q(x,t)))\, dq.$$

We discretize this equation in time using an implicit Euler method, and in space using first-order Lagrangian finite elements. The resulting weak discrete spatial problem can be written as a minimization problem for a strictly convex functional. At each time step, we determine the minimizer of this functional by a monotone multigrid method [Berninger et al., 2011].

For the implementation we used the code used for the numerical examples in [Berninger et al., 2011]. Since vertex-based finite elements were used for the discretization, no changes to the numerical algorithm were needed to also apply it to network grids. Originally, the code used the Dune libraries and the UGGrid grid manager for unstructured grids. Even though the code for [Berninger et al., 2011] was not written with network flow problems in mind at all, it could nevertheless be reused as is, after only a handful of trivial bugfixes. The only changes necessary were replacing the line

C++ code

```
1 typedef UGGrid<2> GridType;
```

by

C++ code

```
1 typedef FoamGrid<2,3> GridType;
```

and adjusting the boundary data specification. This once more proves the point that using the Dune grid interface gives great flexibility, and allows code to do more things than planned by the original authors.

We present an example simulation using an artificially created network grid. The grid, which can be seen in Figure 6, was created by Patrick Laug using the fracture network grid generator described in [Borouchaki et al., 2000]. The grid spans the volume $[-6.5, 6.5]^2 \times [-2.165, 2.165]$ (length and pressure head are given in meters). We assume the network to be filled with a sand-like material with parameters $\theta_m = 0.0458$, $\theta_M = 1$, hydraulic conductivity $K_f = 6.54 \cdot 10^{-5}$ m/s, bubbling pressure $p_b = 0.0726$ m, and $\lambda = 0.694$.

We assume the network to be initially devoid of water, setting $p = -10$ m. Then, water is injected in a unit circle centered at the point $(0, 6.5, 0)$ on the boundary $\partial\Omega \cap \{x_1 = 6.5\}$ with a constant pressure head of $p = 3$ m; no-flow boundary conditions are imposed at the remaining boundary. Figure 7 shows several steps in the evolution of the physical pressure head p. As expected, one can see the fluid entering, and slowly filling almost the entire network. At the end, a steady state is reached, and the pressure head is constant in each connected component of the domain.

4.2 Transport of a therapeutic agent in the microvasculature

The next example demonstrates local grid adaptivity, and the treatment of network bifurcations. We model the propagation of a therapeutic agent for cancer therapy in a network of small blood vessels. To reduce the computational effort, the network of three-dimensional vessels is reduced to a one-dimensional network Ω embedded in a three-dimensional tissue domain $\mathcal{T}$.

We first discuss the one-dimensional model for flow in a blood vessel segment, disregarding any bifurcations. Starting from a three-dimensional blood vessel domain Ξ, we describe blood in the microvasculature as an incompressible Newtonian fluid with viscosity μ and density ϱ governed

© by the authors, 2017

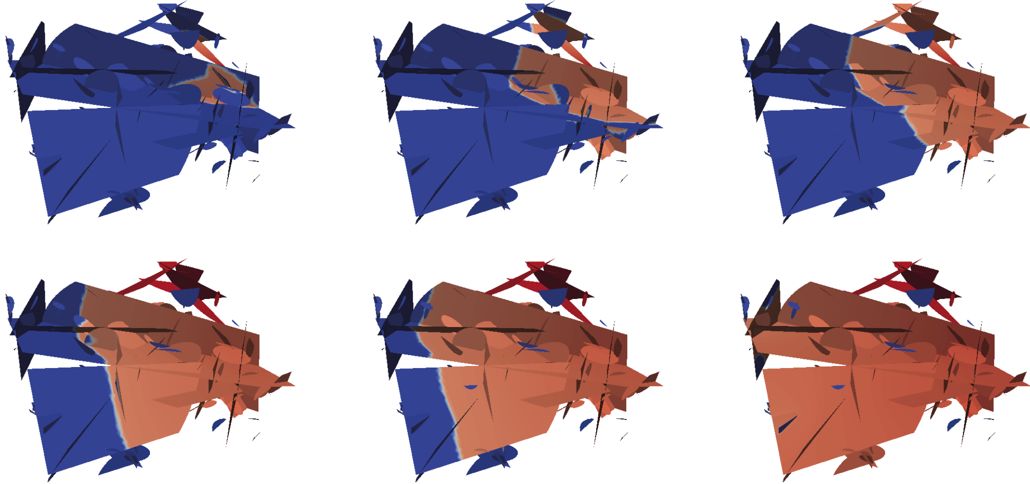


Figure 7: Unsaturated Darcy flow in a fracture network. The color visualizes the physical pressure head p.

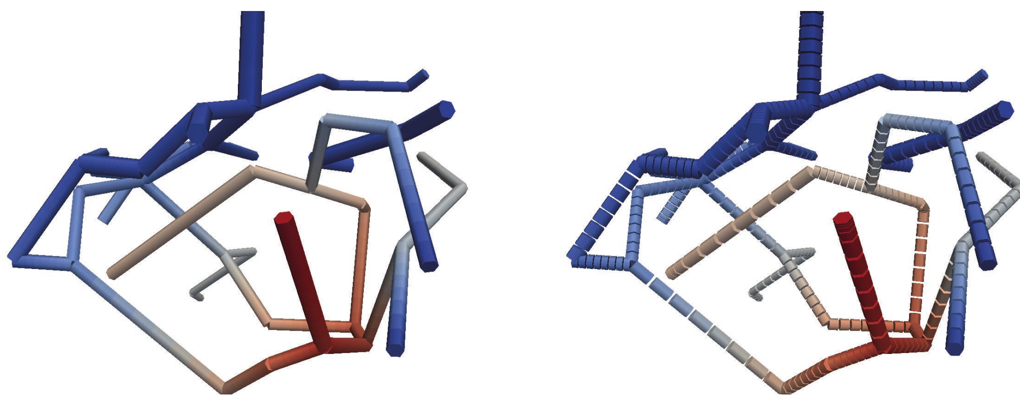


Figure 8: Single-phase two-component flow in a blood vessel network. One-dimensional elements are rendered as tubes. The color visualizes the fluid pressure. Left: stationary pressure p; right: initial element sizes.

 © by the authors, 2017

by the Stokes equation. Further, we assume an axially symmetric blood vessel segment with constant radius R, cross-section area A, the parametrized tangent on the vessel centerline $\lambda(s)$, $\lambda : \mathbb{R} \to \mathbb{R}^3$, $s \mapsto \lambda(s)$, and a rigid vessel wall. Then, with the assumption of constant pressure p in a cross-section and negligible radial velocities v_r the Stokes equations can be integrated over a vessel segment yielding the one-dimensional equation

$$\left.\begin{aligned} \frac{A}{\varrho}\frac{\partial p}{\partial s}\lambda + 2\pi\frac{\mu}{\varrho}(2+\gamma)\bar{v}\lambda = 0 \\ \frac{\partial}{\partial s}\left(\frac{\pi R^4}{2\mu(2+\gamma)}\frac{\partial p}{\partial s}\right) = 0 \end{aligned}\right\} \quad \text{in } \Omega. \tag{3}$$

The parameter γ shapes a power-type axial velocity profile with mean velocity $\bar{v}$, yielding a quadratic velocity profile for $\gamma = 2$ and flatter profiles for $\gamma > 2$. For a detailed derivation in a more general setting, see the reduction of the Navier–Stokes equations to one-dimensional equations in [Quarteroni and Formaggia, 2003]. Equation (3) is a simplified stationary version of the one-dimensional blood flow equations described by Quarteroni and Formaggia [2003], neglecting vessel wall displacement and inertial forces.

Small blood vessels are exchanging mass with the embedding tissue through the vessel wall. The fluid exchange can be modeled by Starling's law, and results in an additional source term. With this modification, (3) becomes

$$\left.\begin{aligned} \frac{A}{\varrho}\frac{\partial p}{\partial s}\lambda + 2\pi\frac{\mu}{\varrho}(2+\gamma)\bar{v}\lambda = 0 \\ \frac{\partial}{\partial s}\left(\frac{\pi R^4}{2\mu(2+\gamma)}\frac{\partial p}{\partial s}\right) - 2\pi R L_p(p-\bar{p}_i) = 0 \end{aligned}\right\} \quad \text{in } \Omega, \tag{4}$$

where L_p is the empirical filtration coefficient dependent on, e.g., the intrinsic permeability and thickness of the membrane, and the viscosity of the interstitial fluid. The source term further depends on the pressure in the surrounding tissue $\bar{p}_i$. For the sake of simplicity, this tissue pressure is subsequently assumed constant. Equation (4) was also used by Cattaneo and Zunino [2014a] in a finite element setting to model coupled vessel-tissue flow processes in a tumor tissue.

The transport of a therapeutic agent is modeled by an advection–diffusion equation using the velocity field calculated by equation (4). Similar to the reduction of the Stokes equations we can reduce the three-dimensional advection–diffusion equation by integration over a vessel segment assuming a constant concentration $c = x\varrho$ on a given cross-section with area $A = \pi R^2$ [D'Angelo, 2007, Cattaneo and Zunino, 2014b]. The transport over the vessel wall can be described by the Kedem–Katchalsky equation [Kedem and Katchalsky, 1958], yielding

$$\frac{\partial(Ac)}{\partial t} + \bar{v}\frac{\partial(Ac)}{\partial s} - D_e\frac{\partial^2(Ac)}{\partial s^2} - 2\pi R\left[L_c(c-\bar{c}_i) + L_p(p-\bar{p}_i)(1-\sigma_c)c\right] = 0 \quad \text{in } \Omega. \tag{5}$$

The last term in (5), accounting for transport across the vessel wall, consists of an advective and a diffusive part where advection is reduced by the reflection coefficient $\sigma_c \in [0,1]$ for larger molecules. Again, we assume the mean tissue concentration $\bar{c}_i$ to be constant.

To model a network of such segments we split the blood vessel network Ω at junctions into pieces yielding a set of vessel segments Ω_i each governed by equations (4) and (5). At each junction we require continuity of pressure

$$p = p_1 = \dots = p_j.$$

Together with these coupling conditions, and boundary conditions on $\partial\Omega$, Equation (4) has a unique solution, which is the stationary pressure field p.

© by the authors, 2017

For the transport at the junctions, we require continuity of concentration

$$c = c_1 = \dots = c_j,$$

and, for Q_i, the volume flux leaving segment Ω_i at the junction, we require mass conservation

$$\sum_{i=1}^{j} Q_i = 0.$$

We discretize equations (4) and (5) in space with a standard finite volume scheme and piecewise linear one-dimensional grid elements. This demands balancing fluxes over the edges of the elements. Assume that we have calculated all element transmissibilities

$$t_i := \frac{\pi R_i^4}{2\mu(2+\gamma)}. \tag{6}$$

Then, the volume flux Q_{ij} from element i to a neighboring element j can be calculated as

$$Q_{ij} = t_{ij}(p_i - p_j) = \frac{t_i t_j}{\sum_{k=0}^{N} t_k}(p_i - p_j). \tag{7}$$

One can see that the transmissibility at branching points is dependent on the transmissibility of all N neighboring elements. Assuming that all element transmissibilities t_i reside in an array `transmissibility`, the calculation of the two-point transmissibilities t_{ij} could look as follows using FoamGrid. Note that only a single loop over the grid entities is necessary to calculate the transmissibilities.

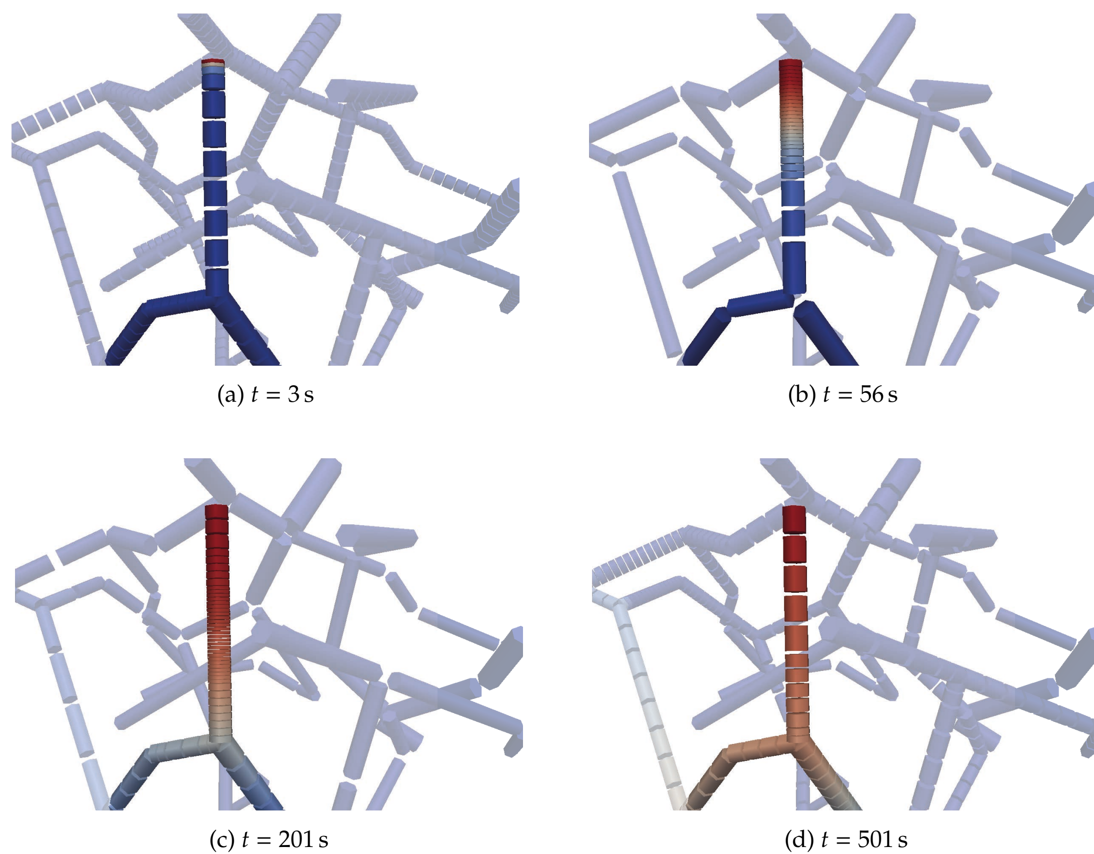
C++ code

```
1  // for each element with index eIdx
2  std::vector<double> tSums(e.subEntities(/*codim=*/ 1), 0.0);
3  std::vector<double> tij;
4  std::vector<std::size_t> neighborFacetMap;
5
6  // loop over all intersections of this element
7  for(const auto& intersection : intersections(foamGridLeafView, element))
8  {
9    if(intersection.neighbor())
10   {
11     std::size_t nIdx = foamGridLeafView.indexSet().index(intersection.outside());
12     tij.push_back(transmissibility[eIdx]*transmissibility[nIdx]);
13     neighborFacetMap.push_back(intersection.indexInInside());
14     tSums[intersection.indexInInside()] += transmissibility[eIdx];
15   }
16   if(intersection.boundary())
17     // boundary treatment ...
18 }
19
20 // compute the two-point transmissibilities
21 for (std::size_t i = 0; i < tij.size(); i++)
22   transmissibilitiesIJ[i] /= tSums[neighborFacetMap[i]];
```

This code works well using the grid interface of `dune-grid-2.4`, even though that interface does not have any provisions for network grids at all (Section 3.1). This is because the grid is one-dimensional. In this case, each intersection always covers entire element facets (i.e., vertices). Therefore, the `indexInInside` method can be used to group intersections.

To reduce numerical diffusion induced by the implicit Euler scheme, the grid can be refined adaptively around the concentration front. Initially, the grid is refined around inflow boundaries,

 © by the authors, 2017



(a) $t = 3\,\mathrm{s}$ (b) $t = 56\,\mathrm{s}$

(c) $t = 201\,\mathrm{s}$ (d) $t = 501\,\mathrm{s}$

Figure 9: Single-phase two-component flow in a blood vessel network. One-dimensional elements are rendered as tubes. The gaps merely exist to better visualize the adaptive grid. The images show the mole fraction x at different simulation times. Note how a concentration wave enters the network from the top, and is tracked by a zone of high grid resolution, even as it goes through a bifurcation point.

© by the authors, 2017

and during the simulation the grid is adapted by a local gradient-based concentration indicator as described in [Wolff, 2013]. This indicator marks an element i for refinement if

$$\frac{\max_i(\Delta c_{ij}) - \Delta c_{\min}}{\Delta c_{\max} - \Delta c_{\min}} \geq \epsilon_r,$$

and for coarsening if

$$\frac{\max_i(\Delta c_{ij}) - \Delta c_{\min}}{\Delta c_{\max} - \Delta c_{\min}} < \epsilon_c,$$

where Δc_{ij} denotes the concentration difference between element i and a neighboring element j. The parameters ϵ_r, $\epsilon_c \in [0,1]$ ($\epsilon_r > \epsilon_c$) are problem dependent. We achieved robust adaptation behavior in our example for $\epsilon_r = 0.3$ and $\epsilon_c = 0.05$. The indicator is evaluated before every time step and marks elements for refinement or coarsening using the `mark` method, see Section 2.3. The adaption of the grid is then handled by FoamGrid. Between the `preAdapt` and `postAdapt` steps, we have to transfer data, i.e., primary variables and spatial parameters from the old grid to the new adapted grid. As can be seen in Figure 9, the refinement scheme works well even around network bifurcations.

We simulate a network of capillaries in rat brain tissue scanned by Motti et al. [1986] and reconstructed as segment network with three-dimensional geometrical information by Secomb et al. [2000]. The network data comprises three-dimensional location of vessel segments, inflow and outflow boundary markers, vessel segment radii, and velocity estimates for each vessel segment. The domain has a bounding box of 150 μm×160 μm×140 μm. The given vessel radii vary between 2 μm and 4.5 μm. The estimated velocities are in a range of 0.5 $\frac{\text{mm}}{\text{s}}$ to 7.5 $\frac{\text{mm}}{\text{s}}$. We choose the blood viscosity $\mu = 3.0 Pa \cdot s$, the filtration coefficient $L_p = 3.33 \cdot 10^{-12} \frac{m}{Pa \cdot s}$, the effective diffusion coefficient over the vessel wall $L_c = 10^{-5} \frac{1}{sm}$, and the diffusion coefficient of the transported agent trail $D_e = 2.93 \cdot 10^{-14} \frac{1}{\text{s}}$, calculated with the Stokes–Einstein radius. We assign the following boundary conditions for the flow problem (4)

$$\begin{aligned} p &= p_D && \text{on } \partial\Omega_{\text{outflow}}, \\ \left[\frac{R^2}{\mu(2+\gamma)} \frac{\partial p}{\partial z} \right] \cdot n &= \bar{v} \cdot n = v_N && \text{on } \partial\Omega_{\text{inflow}}. \end{aligned}$$

The velocities for the inflow segments are set to the estimates provided by Secomb et al. [2000] along with the grid geometry. We simulate the arrival of a therapeutic agent by a Dirichlet boundary condition for equation (5) on a subset $\partial\Omega_c$ of $\partial\Omega_{\text{inflow}}$, namely,

$$\begin{aligned} c &= c_D && \text{on } \partial\Omega_c, \\ c &= 0 && \text{on } \partial\Omega_{\text{inflow}} \setminus \partial\Omega_c. \end{aligned}$$

Specifically, we enforce a mole fraction of $x_D = 10^{-8}$ at one of the inflow vessel segments with a Dirichlet boundary condition. At outflow boundaries, we neglect diffusive fluxes. Note that a full upwind scheme is employed for the concentration, so no further boundary condition for the advective fluxes is necessary at outflow boundaries.

Figure 8 shows the resulting stationary pressure field and the initial element size. In Figure 9, one can see the resulting mole fraction at various times. Note how the local grid refinement follows the steepest gradients, i.e., the transport front. The resulting mole fractions vary from segment to segment due to different radii. This also automatically ensures a finer grid around vessel bifurcations. The one-dimensional elements are depicted as three-dimensional tubes scaled with their respective radius.

 © by the authors, 2017

4.3 Root water uptake and root growth at plant-scale

In environmental and agricultural research fields, models describing root architectures are used to investigate water uptake and root growth behavior of plants [Dunbabin et al., 2013]. In our final example, a one-dimensional network embedded into $\mathbb{R}^3$ is used to describe such a plant root architecture. We simulate water flow through the root network and root growth.

Plant roots can be described by a tree-like network of pipes which consist of xylem tubes [Tyree and Zimmermann, 2002]. We follow the cohesion–tension theory [Tyree, 1997], where the water flow through the root system is governed by the pressure gradient caused by the transpiration rate of the plant above the soil. We assume only vertical flow and no gravity. This leads to a Darcy's law analogy [Doussan et al., 1998]

$$q_x = -K_x \frac{d\psi_x}{dz},$$

where q_x is the water flux in the xylem tubes, ψ_x is the xylem water potential, and K_x is the axial conductance of one root segment.

Water can enter the roots at any point on the xylem tube surfaces, which leads to a volume source term for the one-dimensional network model. For simplicity, we model a single membrane only for the entire pathway of water from soil into the roots. With this assumption, and neglecting osmotic processes, radial water flow q_r into one root segment is defined as

$$q_r = K_r A_r (\psi_S - \psi_x),$$

where K_r is the radial conductivity, i.e., the conductivity of series of tissues from root surface to the xylem. The number $A_r := 2\pi r l$ is the soil–root interface area. The water potential ψ_S at the soil–root interface must be provided. One option is to couple the root system to a Richards equation based soil water flow simulation [Javaux et al., 2008], but for simplicity we simply take ψ_S as a known value.

The continuity equation leads to

$$-\operatorname{div}(q_x(p_x)) = S(p_x) \tag{8}$$

with a solution-dependent source

$$S(p_x) = K_r A_r (\psi_S - \psi_x),$$

where q_x is the only unknown variable. This modeling approach neglects the influence of solutes on water flow, as well as the capacitive effect of the roots, because the amount of water stored in roots is generally small compared to transpiration requirements.

Equation (8) is discretized in space with standard cell-centered finite volumes, piecewise linear one-dimensional grid elements and implemented using the external Dune discretization module DuMu^x [Flemisch et al., 2011], and the FoamGrid grid manager. Flux calculations over edges and branching points of the root network are implemented just as in our previous example.

So far, we have assumed a root network that does not alter its geometry over time. Several algorithms were developed to describe root growth (e.g., [Pagès et al., 2004, Somma et al., 1998, Leitner et al., 2010]). These models define root growth either by a fractal description, or more generically depending on plant specific parameters (root elongation, growth direction, branching density) and surrounding soil properties (soil moisture, soil strength, temperature, nutrients). For simplicity, we model root growth here as an (almost) completely random process. New root branches occur at random time steps and with a small gravity effect only, which makes the roots tend to point downwards. Existing branches grow at the branch tip at random times and without changing directions.

© by the authors, 2017

Our example simulation starts with a simple root grid which consist of one vertical root branch discretized with eight elements (root segments) and no lateral branches (Figure 10). We choose radial and axial conductivity values from [Doussan et al., 1998]. The surrounding relative soil pressure is set to $\psi_S = -2.9429 \cdot 10^{-2} \frac{J}{m^3}$, and the Dirichlet boundary value at the root collar is set to $\psi_{xD} = -1.2 \cdot 10^6 \frac{J}{m^3}$. Parameters and boundary conditions do not change with time.

In every time step, new root elements are created and either added to an existing lateral branch or to the main branch. An element-based indicator based on simulation time and random factors decides whether a new branch is added at one of the element's vertices. The `Indicator` class also computes the coordinates of the newly inserted point that is the second vertex of the new element. The coordinates depend again on simulation time, random factors, and the branch orientation in $\mathbb{R}^3$. Our growth step calculation in DuMu^x using FoamGrid looks as follows.

C++ code

```
1  template<class Indicator>
2  void growGrid(Indicator& indicator, Variables& vars)
3  {
4    // (1) calculate indicator for each element
5    indicator.calculateIndicator();
6
7    // (2) insert elements according to the computed indicator
8    insertElements(indicator);
9
10   // (3) Put variables in a persistent map
11   storeVariables(vars);
12
13   // (4) Grow grid
14   grid->grow();
15
16   // (5) Resize and (re-)construction of variables
17   vars.resize(foamGridLeafView.size(0));
18   reconstructsVariables(vars);
19
20   // (6) delete isNew markers in grid
21   grid->postGrow();
22 }
```

The `vars` container contains all primary variables and spatial parameters defined on the root segments. The `Indicator` class is a template parameter that can be easily exchanged, to allow different root growth algorithms. In Step (2), the new elements are inserted. New root segments must be connected to the old grid. The implementation of the `insertElements` method could looks as follows:

C++ code

```
1  template<class Indicator>
2  void insertElements(const Indicator& indicator)
3  {
4    // iterate over all grid elements (root segments)
5    for (const auto& element : elements(foamGridLeafView))
6    {
7      // find elements that will get a new neighbor
8      if (indicator.willGrow(element))
9      {
10       // get the new elements vertices from the indicator
11       std::size_t vIdx0 =
             grid->insertVertex(indicator.getNewVertexCoordinates(element));
12       // get index of the existing element vertex the new element will be connected to
13       std::size_t vIdx1 = indicator.getConnectedVertex(element);
14       // insert new element with the two vertex indices
15       grid->insertElement(element.type(), {vIdx0, vIdx1});
16     }
17   }
```

 © by the authors, 2017

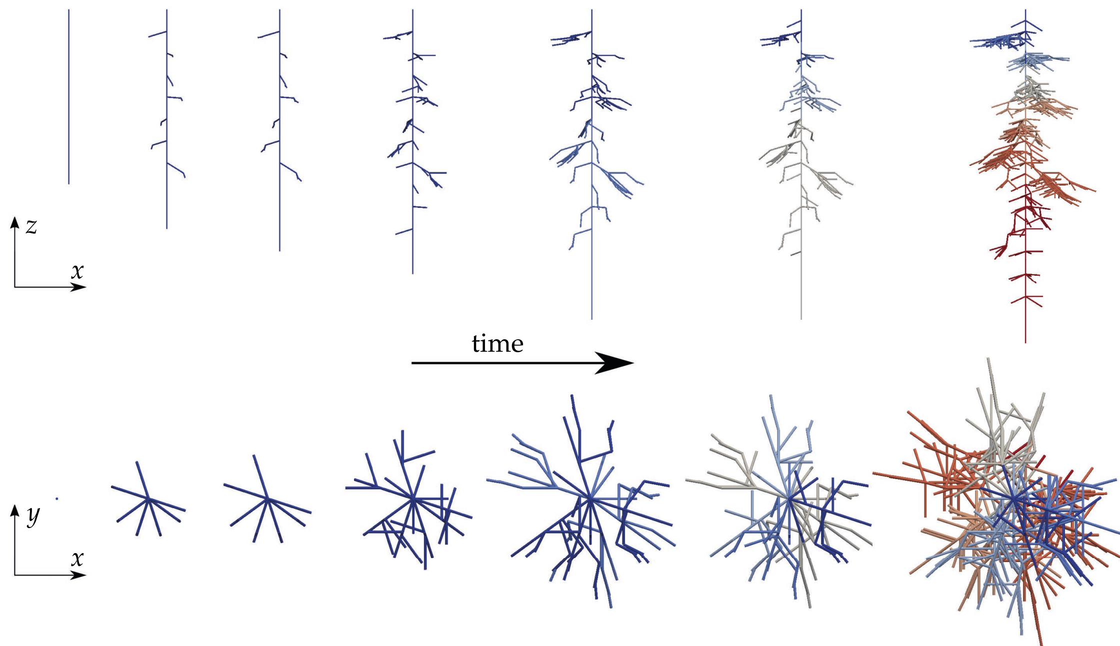


Figure 10: Growth of a root system, shown in a lateral (top) and an axial (bottom) view. The color represents the pressure inside the roots.

```
18 }
```

The primary variables and spatial parameters of our physical problem are stored in Step (3), before the actual growth step (4). We update the sizes of the variables and parameter vectors since the total number of degrees of freedom has changed due to the growth step (5). In addition, values for the new elements have to be computed. In our case, new root segments inherit the primary variables and the spatial parameters from its preceding neighbor element. Root tips, in particular new root tips, are always assigned Neumann no-flow boundary conditions. At the end, the `postGrow` method is called, which deletes the `isNew` markers (Step (6)).

Figure 10 shows the root network and the pressure distribution inside the roots for several time steps. The total root water uptake (transpiration demand) of the plant, defined by the Dirichlet boundary condition, does not change during the growing period. Thus, the pressure inside the plant changes since the water uptake of the plant is distributed to more and more roots segments.

References

H. Alt and S. Luckhaus. Quasilinear elliptic–parabolic differential equations. *Math. Z.*, 183: 311–341, 1983.

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.

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.

P. Bastian, G. Buse, and O. Sander. Infrastructure for the coupling of Dune grids. In *Proc. of ENUMATH 2009*, pages 107–114. Springer, 2010.

© by the authors, 2017

J. Bear. *Dynamics of Fluids in Porous Media*. Dover Publications, 1988.

H. Berninger, R. Kornhuber, and O. Sander. Fast and robust numerical solution of the Richards equation in homogeneous soil. *SINUM*, 49(6):2576–2597, 2011.

H. Borouchaki, P. Laug, and P. George. Parametric surface meshing using a combined advancing-front – generalized-Delaunay approach. *Int. J. Numer. Meth. Eng.*, 49(1–2):233–259, 2000.

A. Bressan, S. Čanić, M. Garavello, M. Herty, and B. Piccoli. Flows on networks: recent results and perspectives. *EMS Surv. Math. Sci.*, 1(1):47–111, 2014.

L. Cattaneo and P. Zunino. Computational models for fluid exchange between microcirculation and tissue interstitium. *Networks and Heterogeneous Media*, 9(1):135–159, 2014a.

L. Cattaneo and P. Zunino. A computational model of drug delivery through microcirculation to compare different tumor treatments. *International Journal for Numerical Methods in Biomedical Engineering*, 30(11):1347–1371, 2014b.

C. D'Apice, S. Göttlich, M. Herty, and B. Piccoli. *Modeling, simulation, and optimization of supply chains*. SIAM, 2010.

C. Doussan, L. Pages, and G. Vercambre. Modelling of the Hydraulic Architecture of Root Systems: An Integrated Approach to Water Absorption—Model Description. *Annals of Botany*, 81(2):213–223, 1998. doi: 10.1006/anbo.1997.0540.

V. M. Dunbabin, J. A. Postma, A. Schnepf, L. Pagès, M. Javaux, L. Wu, D. Leitner, Y. L. Chen, Z. Rengel, and A. J. Diggle. Modelling root–soil interactions using three-dimensional models of root growth, architecture and function. *Plant and Soil*, 372:93–124, 2013. doi: 10.1007/s11104-013-1769-y.

G. Dziuk and C. M. Elliott. Finite elements on evolving surfaces. *IMA J. Numer. Anal.*, 27:262–292, 2007a.

G. Dziuk and C. M. Elliott. Surface finite elements for parabolic equations. *J. Comput. Math.*, 25 (4):385–407, 2007b.

G. Dziuk and C. M. Elliott. Finite element methods for surface PDEs. *Acta Numerica*, 22:289–396, 2013.

C. D'Angelo. Multiscale modelling of metabolism and transport phenomena in living tissues. *Bibliotheque de l'EPFL, Lausanne*, 2007.

C. Engwer and S. Müthing. Concepts for flexible parallel multi-domain simulations. In T. Dickopf, M. Gander, L. Halpern, R. Krause, and L. Pavarino, editors, *Domain Decomposition Methods in Science and Engineering XXII*, volume 104 of *Lecture Notes in Computational Science and Engineering*, pages 187–195. Springer, 2016.

B. Flemisch, M. Darcis, K. Erbertseder, B. Faigle, A. Lauser, K. Mosthaf, P. Nuske, A. Tatomir, M. Wolff, and R. Helmig. DuMuX : DUNE for Multi-{ Phase, Component, Scale, Physics, . . . } Flow and Transport in Porous Media. *Advances in Water Resources*, 34:1102–1112, 2011.

M. Garavello and B. Piccoli. *Traffic flow on networks*. American Institute of Mathematical Sciences (AIMS), 2006.

C. Geuzaine and F. Remacle. *Gmsh Reference Manual*, 2015. URL www.geuz.org/gmsh/doc/texinfo/gmsh.pdf.

S. Gross and A. Reusken. *Numerical Methods for Two-phase Incompressible Flows*. Springer, 2011.

 © by the authors, 2017

M. Javaux, T. Schröder, J. Vanderborght, and H. Vereecken. Use of a Three-Dimensional Detailed Modeling Approach for Predicting Root Water Uptake. *Vadose Zone Journal*, 7(3):1079, 2008. doi: 10.2136/vzj2007.0115.

O. Kedem and A. Katchalsky. Thermodynamic analysis of the permeability of biological membranes to non-electrolytes. *Biochimica et biophysica Acta*, 27:229–246, 1958.

S. Lang, V. J. Dercksen, B. Sakmann, and M. Oberlaender. Simulation of signal flow in 3d reconstructions of an anatomically realistic neural network in rat vibrissal cortex. *Neural Networks*, 24(9):998–1011, 2011.

D. Leitner, S. Klepsch, G. Bodner, and A. Schnepf. A dynamic root system growth model based on L-Systems. *Plant and Soil*, 332:177–192, Jan. 2010. ISSN 0032-079X. doi: 10.1007/s11104-010-0284-7.

M. McClure, M. Babazadeh, S. Shiozawa, and J. Huang. Fully coupled hydromechanical simulation of hydraulic fracturing in three-dimensional discrete fracture networks. In *SPE Hydraulic Fracturing Technology Conference, 3–5 February, The Woodlands, Texas, USA*, 2015. doi: http://dx.doi.org/10.2118/173354-MS.

M. W. McClure and R. N. Horne. *Discrete Fracture Network Modeling of Hydraulic Stimulation – Coupling Flow and Geomechanics*. Springerbriefs in Earth Sciences. Springer, 2013.

E. D. Motti, H.-G. Imhof, and M. G. Yasargil. The terminal vascular bed in the superficial cortex of the rat: an SEM study of corrosion casts. *Journal of Neurosurgery*, 65(6):834–846, 1986.

S. Nammi, P. Myler, and G. Edwards. Finite element analysis of closed-cell aluminium foam under quasi-static loading. *Materials and Design*, 31:712–722, 2010.

I. Nitschke, A. Voigt, and J. Wensch. A finite element approach to incompressible two-phase flow on manifolds. *J. Fluid Mech.*, 708:418–438, 2012.

M. A. Olshanskii, A. Reusken, and J. Grande. A finite element method for elliptic equations on surfaces. *SIAM J. Numer. Anal.*, 47(5):3339–3358, 2009.

L. Pagès, G. Vercambre, and J. Drouet. Root Typ: a generic model to depict and analyse the root system architecture. *Plant and Soil*, 258:103–119, 2004.

A. Quarteroni and L. Formaggia. Mathematical Modelling and Numerical Simulation of the Cardiovascular System. In *Modelling of Living Systems*, Handbook of Numerical Analysis Series. EPFL, 2003.

S. Reuther and A. Voigt. The interplay of curvature and vortices in flow on curved surfaces. *SIAM Multiscale Model. Simul.*, 13(2):632–643, 2015.

M. E. Rognes, D. A. Ham, C. J. Cotter, and A. T. T. McRae. Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2. *Geosci. Model Dev.*, 6:2099–2119, 2013.

A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software. The Finite Element Toolbox ALBERTA*, volume 42 of *LNCSE*. Springer, 2005. URL alberta-fem.de.

T. Secomb, R. Hsu, N. Beamer, and B. Coull. Theoretical simulation of oxygen transport to brain by networks of microvessels: effects of oxygen supply and demand on tissue hypoxia. *Microcirculation*, 7(4):237–247, 2000.

F. Somma, J. W. Hopmans, and V. Clausnitzer. Transient three-dimensional modeling of soil water and solute transport with simultaneous root growth, root water and nutrient uptake. *Plant and Soil*, 202:281–293, 1998.

© by the authors, 2017

A. V. T. Witkowski, R. Backofen. The influence of membrane bound proteins on phase separation and coarsening in cell membranes. *Phys. Chem. Chem. Phys.*, 14:14509–14515, 2012.

M. T. Tyree. The Cohesion-Tension theory of sap ascent : current controversies. *Journal of Experimental Botany*, 48(315):1753–1765, 1997.

M. T. Tyree and M. H. Zimmermann. *Xylem Structure and the Ascent of Sap*, volume 12. Springer, 2002. ISBN 9783540433545. doi: 10.1016/0378-1127(85)90081-7.

S. Vey and A. Voigt. AMDiS: adaptive multidimensional simulations. *Comput. Visual. Sci.*, 10: 57–67, 2007.

M. Wolff. *Multi-scale modeling of two-phase flow in porous media including capillary pressure effects*. PhD thesis, Universität Stuttgart, 2013.

 © by the authors, 2017

Index

Editorial: Proceedings of the 3rd Dune User Meeting
Markus Blatt / Bernd Flemisch / Oliver Sander

Using DUNE-ACFEM for Non-smooth Minimization of Bounded Variation Functions // Martin Alkämper / Andreas Langer

The DUNE-FEM-DG module
Andreas Dedner / Stefan Girke / Robert Klöfkorn / Tobias Malkmus

Asynchronous evaluation within parallel environments of coupled finite and boundary element schemes for the simulation of multiphysics problems // Andreas Dedner / Alastair J. Radcliffe

The interface for functions in the dune-functions module
Christian Engwer / Carsten Gräser / Steffen Müthing / Oliver Sander

The DUNE-DPG library for solving PDEs with Discontinuous Petrov–Galerkin finite elements
Felix Gruber / Angela Klewinghaus / Olga Mula

Using DUNE-FEM for Adaptive Higher Order Discontinuous Galerkin Methods for Two-phase Flow in Porous Media // Birane Kane

System testing in scientific numerical software frameworks using the example of DUNE // Dominic Kempf / Timo Koch

FunG – Automatic differentiation for invariant-based modeling
Lars Lubkoll

Extending DUNE: The dune-xt modules
Tobias Leibner/ René Milk / Felix Schindler

The Dune FoamGrid implementation for surface and network grids
Oliver Sander / Timo Koch / Natalie Schröder / Bernd Flemisch

UNIVERSITÄT HEIDELBERG
ZUKUNFT
SEIT 1386

ISBN 978-3-946531-61-6