A Flexible Framework for Multi Physics and Multi Domain PDE Simulations

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Steffen Müthing

aus Olsberg

Hauptberichter:	Prof. Dr. Peter Bastian
Mitberichter:	Prof. Dr. Thomas Ertl

Tag der mündlichen Prüfung: 20. Februar 2015

Institut für Parallele und Verteilte Systeme
der Universität Stuttgart

2015

Abstract

Many important problems in physics and engineering like fluid dynamics and continuum mechanics are modeled using partial differential equations. These problems can typically not be solved directly, but have to be approximated numerically, a challenging process at both the mathematical and the computer science level.

In this work, we present a novel set of software components that facilitate the creation of simulation programs for multi domain partial differential equation problems. We identify the implementation challenges related to the coupling of multiple spatial domains and their attached physical problems and develop a mathematical framework of clearly defined building blocks that can be used to compose a multi domain problem by combining single physics building blocks (which are typically already well understood by application scientists) with additional components that describe the interactions between those subproblems.

We introduce an open source software implementation of these mathematical concepts on top of the well-established DUNE numerics framework. This implementation consists of two major parts: a mechanism to subdivide any existing DUNE mesh into multiple subdomains, and a set of extensions to the high-level partial differential equation toolbox solver PDELab, which make the components of our mathematical framework available within its solvers. Our overall design enables application-level scientists to reuse existing code blocks from single-physics simulations and combine them to solve new multi domain problems.

This new functionality is heavily based on PDELab's recursive tree representation of product function spaces; we replace the internal ad-hoc implementation of these trees with a new C++ library for statically defined, template-based trees of objects. As multi domain problems typically require structured linear algebra solvers that exploit domain decomposition approaches, we develop a mathematical framework for describing the structure of the vectors and matrices generated during the assembly of a partial differential equation problem based on the structure of the underlying function spaces. This framework is implemented in PDELab; it is based on a tree transformation mechanism provided by our tree library.

We demonstrate the versatility of our multi domain simulation components and their impact on developer productivity by means of two model examples; our ultimate goal of simplifying the development of real-world applications is shown by a description of the impact of our software on several external research projects. Finally, we measure the performance impact of our extensions on the existing DUNE framework and discuss the mitigation measures we implemented to reduce any existing performance penalties.

German Abstract
—Zusammenfassung—

Sehr viele Probleme der Physik und Ingenieurswissenschaften wie zum Beispiel Fluiddynamik und Kontinuumsmechanik werden durch Modelle beschrieben, die auf partiellen Differentialgleichungen basieren. Da es von wenigen Ausnahmen abgesehen nicht möglich ist, diese Probleme analytisch zu lösen, wird die Lösung typischerweise numerisch approximiert, wobei die Entwicklung entsprechender Simulations-Software weitreichende Kenntnisse auf mathematischer Ebene wie auch im Feld der Informatik erfordert.

In dieser Arbeit stellen wir eine neue Sammlung von Software-Komponenten für die Simulation von sogenannten Mehrgebietsproblemen vor, die aus mehreren partiellen Differentialgleichungen auf unterschiedlichen räumlichen Gebieten bestehen. Wir arbeiten die Schwierigkeiten heraus, die aus der computertechnischen Umsetzung der auf Ebene der Gleichungen und der diskretisierten Simulationsgebiete bestehenden Kopplungen resultieren. Basierend auf dieser Analyse entwickeln wir ein mathematisches Gerüst aus wohldefinierten Komponenten, die es uns erlauben, bereits für die einzelnen Bestandteile unseres Problems bestehende Modelle zu kombinieren. Komplexe Modelle für Multiphysik-Probleme können dann konstruiert werden, indem diese Teilprobleme um zusätzlicher Kopplungsbausteine ergänzt werden, die die Interaktionen zwischen den einzelnen Teilproblemen beschreiben.

Wir entwickeln eine Open-Source Software-Umsetzung dieser Konzepte auf Basis des etablierten DUNE Numerik-Frameworks, welche aus zwei zentralen Komponenten besteht: einem Erweiterungsmechanismus, der es uns erlaubt, beliebige DUNE-Gitter in mehrere Teilgebiete zu zerlegen, und einem Satz von Erweiterungen für das hochabstrahierte Löserpaket PDELab, welche unsere mathematischen Abstraktionen auf Software-Komponenten umsetzen und für Programme auf PDELab-Basis verfügbar machen. Unsere Bibliothek ist so gestaltet, dass Anwender existierende Simulationen für einfache Probleme ohne Änderungen am Quellcode wiederverwenden und kombinieren können, um neue Mehrgebietsprobleme zu lösen.

Die Umsetzung dieser Anforderungen baut in großem Maße auf der rekursiven Baum-Repräsentation von Mehrkomponenten-Funktionenräumen in PDELab auf; um deren Funktionalität für unsere Zwecke erweitern zu können, ersetzen wir die Ad-hoc-Implementierung dieser Bäume durch eine neuentwickelte, leistungsfähige Bibliothek für statisch definierte, template-basierte Bäume aus heterogenen Objekten.

Mehrgebietsprobleme erfordern typischerweise spezielle Löser, die auf die Problemstruktur angepasst sind, z.B. Gebietszerlegungsverfahren. Um derartige Löser effizient zu implementieren, sollte sich die Problemstruktur in den Blockstrukturen der assemblierten Vektoren und Matrizen widerspiegeln. Zu diesem Zweck entwicklen wir ein allgemeines Konzept, um die Umsortierung und Blockung von

Freiheitsgraden zu beschreiben, und implementieren dieses als Teil von PDELab, wiederum basierend auf unserer Bibliothek für heterogene Bäume.

Wir demonstrieren die Mächtigkeit der gesamten Mehrgebietssimulations-Bibliothek und die erzielte Reduktion des Entwicklungsaufwands anhand zweier Modellbeispiele. Insbesondere Simulationen von realen, applikationsgetriebenenen Problemen können durch unsere Software deutlich vereinfacht werden, was wir durch einen kleinen Überblick über externe Forschungsprojekte, die auf unserer Software aufbauen, untermauern. Schließlich evaluieren wir den Einfluss unserer Erweiterungen auf die Leistung der unterliegenden DUNE-Komponenten und diskutieren die von uns implementierten Maßnahmen zur Minimierung von Leistungsverlusten.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations and Acronyms

ADL	argument-dependent lookup
API	Application Programming Interface
BCRS	block compressed row storage, extension of CRS
CFD	computational fluid dynamics
CPU	central processing unit
CRS	compressed row storage, a sparse matrix storage scheme
CUDA	Compute Unified Device Architecture developed by NVIDIA
DG	Discontinuous Galerkin
DNAPL	dense non aqueous phase liquid
DOF	degree of freedom
DSL	domain-specific language
DUNE	Distributed and Unified Numerics Environment
FE	Finite Element
FV	Finite Volume
FEM	Finite Element Method
FDM	Finite Difference Method
FSI	fluid structure interaction
FVM	Finite Volume Method
GLSL	OpenGL shading language
GPGPU	general purpose computating on graphics processing units
GPU	graphics processing unit
HPC	high performance computing
I/O	input / output
ISO	International Standards Organization
ISTL	Iterative Solver Template Library
JSC	Jülich Supercomputing Centre
LA	linear algebra
MC	Monte Carlo
MIMD	multiple instruction, multiple data
MPI	Message Passing Interface (MIMD processing standard and library)
ODE	ordinary differential equation
ODR	one definition rule
OpenCL	open compute language
OpenGL	open graphics libray
OpenMP	open multi-processing

PDE partial differential equation
RAM random access memory
SDE stochastic differential equation
SFINAE Substitution Failure Is Not An Error
SIMD single instruction, multiple data
SIMT single instruction, multiple threads
SISD single instruction, single data
SIPG Symmetric Interior Penalty Galerkin, a particular, symmetric DG method
STL Standard Template Library
TMP Template Meta Program
VTK Visualization ToolKit (API and file format of the open-source visualization tool ParaView)
XFEM Extended Finite Element Method

Units

GiB gibibyte, 2^{30} bytes
KiB kibibyte, 2^{10} bytes
MiB mebibyte, 2^{20} byte
TiB tebibyte, 2^{40} bytes

List of Symbols

Symbol	Explanation
α_T^{vol}	integration kernel for volume contributions on cell T to discrete residual
α_τ^{bnd}	integration kernel for boundary contributions on outer boundary intersection τ to discrete residual
$\alpha_\tau^{\text{skel}}$	integration kernel for skeleton contributions on internal intersection τ to discrete residual
$\boldsymbol{b}$	convection velocity
$\mathcal{B}$	tree
c	reaction rate or codimension
C	coupling
d	spatial dimension
$\boldsymbol{D}$	diffusion coefficient
$\mathcal{D}(\varphi)$	DOFIndex of basis function φ
$\mathbb{D}$	symmetric deformation tensor
ϵ_0	vacuum permittivity
$\mathcal{F}$	tree transformation
g	gravitational acceleration
$g(T, l)$	map from local enumeration of basis functions on cell T to global enumeration
Γ	surface, part of $\partial\Omega$
Γ_C	coupling surface
h	cell size
$\mathcal{I}$	index tuple
$\mathbb{I}$	identity matrix
k_B	Boltzmann constant
$k_{r,\alpha}$	relative permeability of phase α
$\boldsymbol{K}$	absolute permeability tensor
μ_T	geometry transformation from $\hat{\Omega}_T$ to global coordinates
$n(T)$	number of basis functions associated with T
ν	viscosity
$\boldsymbol{n}$	normal unit vector
$m(u, v)$	residual form for temporal part of instationary PDEs
$\mathcal{M}$	discrete residual operator for temporal part
Ω	spatial domain
$\partial\Omega$	surface of a spatial domain
$\hat{\Omega}_T$	reference element of grid cell T
p	maximum degree of a polynomial
p_α	pressure of phase α

Symbol	Explanation
p_c	capillary pressure
P	subproblem
φ	basis function
Φ	basis
$\mathbb{P}_k^d$	space of polynomials up to degree k in $\mathbb{R}^d$
q	mesh predicate for subproblems
$r(u, v)$	residual form
$r^h(u, v)$	discrete residual form
R_T	restriction operator from global DOF vector $\boldsymbol{u}$ to restricted version $\boldsymbol{u}^T$ on cell T
$\mathcal{R}$	discrete residual operator
ρ	density
s	subdomain
s_α	saturation of phase α
S	subdomain set
$\mathcal{T}$	tessellation
$\mathcal{T}_\mathcal{I}$	restricted tessellation for subspace index tuple $\mathcal{I}$
T	tessellation cell
$\tau^{(c)}$	tessellation entity with codimension c
$\mathbb{T}$	stress tensor
$\boldsymbol{u}$	DOF vector with basis function coefficients
U	ansatz space
$\boldsymbol{v}$	velocity
V	test space or general function space
V^h	discrete space on mesh with cell size h
$V_\mathcal{I}$	subspace of V induced by subspace index tuple $\mathcal{I}$
$V_{\mathcal{I},\mathcal{T}_\mathcal{I}}$	restricted subspace of V for subspace index tuple $\mathcal{I}$
$\hat{\boldsymbol{x}}$	spatial position in reference element coordinates
$\{u\}$	$\frac{1}{2}(u^+ + u^-)$, average of function value on discontinuity (e.g. internal intersection)
$[u]$	$(u^+ - u^-)$, jump of function value across discontinuity (e.g. internal intersection)

CHAPTER 1

Introduction

1.1 Motivation and Scope

Many interesting problems in the natural sciences and in engineering today involve the simultaneous examination of multiple physical models which interact by means of a set of coupling conditions, for example climate models that include transport processes between water in the atmosphere and the soil or biological models of blood vessels and the surrounding cells. For a large part of these problems, the model can be written as a system of partial differential equations (PDEs). Except for trivial example problems, it is in general not possible to obtain an analytical solution for such a system. In order to investigate those problems, the underlying PDEs are discretized by means of the Finite Element Method (FEM) or similar approaches. Ultimately, this leads to a (possibly nonlinear) system of algebraic equations, which then has to be solved numerically.

Multi (physics, domain, scale) simulations are an essential tool for understanding such complex processes and / or being able to simulate those processes. Going "multi" can be driven either by the structure of the problem or by the need to combine multiple models of different complexity in order to reduce the overall computational cost of the simulation.

As a good example for a problem that by its very nature combines multiple physical models and multiple domains, consider the field of fluid structure interaction (FSI), which is concerned with the interactions between a deformable structure (e.g. an airplane) and the surrounding fluid. It has important real world applications e.g. in aeronautical engineering, where it is used to minimize aircraft drag by shape optimization and to identify weak points in the aircraft structure (structural failure analysis). Figure 1.1 depicts a result of simulating a Mach 7 air flow around a

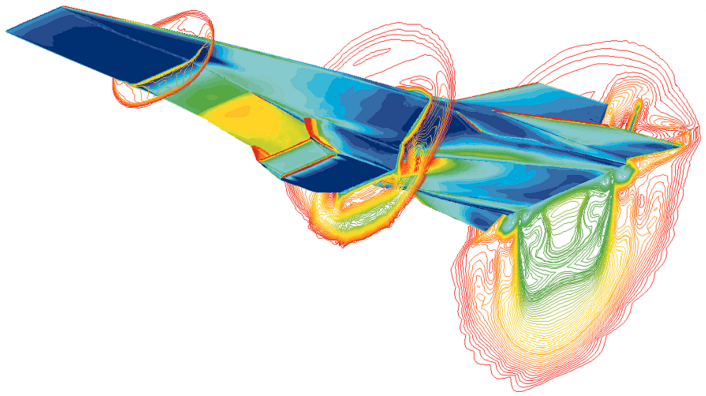


Figure 1.1 — FSI simulation of the X-43A experimental aircraft at Mach 7. Via NASA[1] [Public Domain].

hypersonic airplane and shows both velocity information for the air flow and induced stresses in the aircraft structure. Problems of this kind pose a wide variety of challenges, ranging from the moving geometry due to deformation of the structure to the enormous difficulties in solving the resulting nonlinear algebraic equation systems, for which there are two general approaches: One class of methods splits the system using domain decomposition approaches (Steklov-Poincaré operators, cf. [108, 123]) as performed by e.g. Yang [125], making the individual problems far easier to solve. On the other hand, the subproblem solvers then need to be coupled using some kind of iteration scheme. Creating an overall simulation with predictable convergence is very hard in the general case. An alternative school of thought tries to avoid these problems by creating a monolithic solver as proposed in [122, 44], where the authors argue that for complicated setups, convergence is impossible to guarantee otherwise. However, existing solvers for those fully coupled problem formulations do not yet scale to realistic problem sizes.

Another reason for developing a multi domain simulation arises from applications which can be modeled in multiple ways, each model offering a different trade-off between computational cost and accuracy. Multi domain simulations then make it possible to choose different models in different areas of the overall spatial domain. This approach can for example be exploited when investigating the injection of CO_2 into groundwater as depicted in Figure 1.2: Initially, the CO_2 will only be present in a small region, and a simulation can greatly benefit from only modeling the oil-water interaction in that area while using a simpler water-only model everywhere else.

Looking at these two prototypical applications of multi domain models, it becomes apparent that the development of simulations for such problems is a very complex

[1] http://www.dfrc.nasa.gov/Gallery/Photo/X-43A/HTML/ED97-43968-1.html

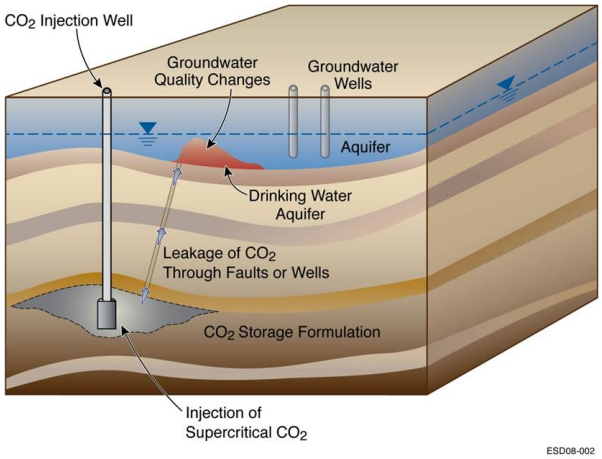


Figure 1.2 — Sketch of a CO_2 injection simulation. Via Lawrence Berkeley National Laboratory[2] [Public Domain].

task that requires a high level of proficiency in the three distinct fields of model development (typically done by application scientists), mathematics (creation of stable and efficient discretizations and solvers), and computer science (implementation complexity and efficiency). In our thesis we develop software tools that abstract away a large part of the computer science related challenges, greatly simplifying the implementation part of simulation development for multi domain problems and making the development of such simulations much more accessible to mathematicians and application scientists.

1.2 Related Work

The numerical solution of PDEs is a field of science that has been expanding for the last several decades. Traditionally, there has been a very pronounced divide between method development, which is focused on analytic improvements and typically validates new algorithms by means of small, purpose-built numerical examples (often built in an ad-hoc fashion in agile environments like MATLAB), and the application side, often driven by physicists and engineers rather than mathematicians, who tend to develop large, highly specialized simulation codes focused on their specific application. These codes tend to be tuned very well, but are often stuck with outdated mathematical methods, as their monolithic design makes it very hard to adapt important algorithms.

While not limited to this goal, scientific computing tries to bridge that gap by developing highly flexible algorithms and software that make it possible to quickly develop prototypical applications which require a large amount of control over

[2] http://esd.lbl.gov/research/programs/gcs/projects/storage_resources/co2intrusion.html

the simulation process, while at the same time providing the necessary infrastructure to extend this core simulation with additional functionality like alternative discretizations, additional solvers and support for parallel computations.

In general, a PDE simulation can be split into two distinct phases: *problem assembly*, which involves computing the vectors and matrices which describe the algebraic problem posed by the discretized PDE model, and *problem solution*, which is occupied with the solution of that (potentially nonlinear) algebraic problem.

1.2.1 PDE Assembly Frameworks

Over the years, the repetitive nature of Finite Element (FE) problem assembly has given rise to a large number of software packages that automate this process to varying degrees. These packages range from mostly grid-centered approaches (e.g. ALBERTA [113, 112, 2], ALUGrid [25, 4]) over more integrated approaches (e.g. deal.II [13], UG [14, 17, 117], diffpack [80, 36]) up to highly integrated frameworks that employ domain-specific languages (DSLs) as a high-level interface for specifying the model equations (sundance [83], freefem [65, 66], FEniCS [82]). This thesis is also based on an established numerics framework, the Distributed and Unified Numerics Environment (DUNE), a modular collection of heavily template-reliant C++ modules based around the idea of a fine-grained interface layer that provides a unified front end for existing software and which can be optimized away at compilation time to provide a good combination of flexibility, versatility and performance.

Unfortunately, the majority of these established simulation frameworks are mostly focused on solving problems on a common domain. Moreover, the standard approach to modeling these subdomains in a simulation has been to employ a distinct grid for each subdomain, mostly motivated by the desire to reuse existing software packages for the subproblems. One major drawback of this approach is the great amount of manual setup work required during the creation of the individual grids, especially in the case of interface problems with complex boundary geometries, where both sides of the interface need to match the common geometric shape of the interface as closely as possible. Moreover, any calculation of coupling conditions, whether on a lower-dimensional interface or on overlapping subdomains, requires a conforming matching of the underlying grid discretizations, which for distinct grids will often necessitate the very expensive calculation of a common sub-tessellation. Examples of such software packages which are capable of handling inter-program mesh and DOF transfer and coordinating a weakly coupled solution scheme include SIERRA [45] and MpCCI [89]. The more recent PreCICE library project [107, 116] is trying to couple existing single-physics black box codes at high performance computing (HPC) scale.

In this thesis, we will take a different approach: Our framework assumes control over the complete simulation stack including all subproblem solvers to create a tightly integrated application, which allows us to investigate both monolithic and

weakly coupled solvers and sidesteps the problem of having to match opaque mesh topologies. Due to this high integration level, our solution is much more amenable to the type of exploratory software development typically associated with method development at both the model and numerics level.

1.2.2 Linear Algebra Libraries

The second major part of any PDE simulation involves the solution of the assembled linear or nonlinear algebraic systems, a problem setting with characteristics that are markedly different from the assembly; in particular, it involves a much more constrained and well-defined set of data structures and algorithms (vectors and matrices as well as linear/nonlinear solvers and preconditioners). It is thus much easier to use existing linear algebra (LA) libraries in custom simulations, and many software packages feature a dedicated interface that allows their users to choose between different LA packages and to pick the best one for the current problem.

This greater tendency towards implementation reuse has also led to a concentration on a small number of larger software projects. The two largest efforts in this field are PETSc [12, 11, 10] and Trilinos [68, 115]. In addition to a number of different sparse matrix implementations, those packages contain a wide variety of linear solvers, advanced preconditioners (including multigrid preconditioners), nonlinear solvers and ordinary differential equation (ODE) solvers for time discretization and provide support for parallelization via Message Passing Interface (MPI), open multi-processing (OpenMP) etc. Many of the more specialized components are actually developed externally, with PETSc mostly bundling external projects and wrapping them in a PETScified Application Programming Interface (API), while Trilinos takes a more integrated approach of a package-based overall framework where the individual components are usually built on top of a common core that provides the basic data structures. However, its use is complicated because it contains several different (and incompatible) vector and matrix implementations.

Apart from those two very large projects (both funded by US national labs), there are two more relatively well-known packages that also see comparatively widespread usage in the form of Eigen [60] and the Matrix Template Library (MTL4) [56, 85]. Eigen right now lacks support for large-scale parallelism based on MPI, while MTL4 is split into a non-parallelized open source core and a commercially licensed part that adds support for parallel computations. Both libraries are written in advanced, heavily templated C++, which starkly contrasts with PETSc (C-based) and – to a lesser extent – Trilinos. They employ operator overloading to provide a very natural syntax for elementary operations on vectors and matrices and use expression templates and lazy evaluation for efficiently evaluating those expressions. Those features make it significantly easier to directly implement many standard solvers from the mathematical descriptions of their algorithms, which partially makes up for the fact that they include far fewer solver components than the two larger frameworks.

Finally, there is the uBLAS library [121], which is part of Boost [114], but which has not seen any development lately and is missing built-in solvers and support for parallelism.

DUNE (more specifically, its solver toolbox PDELAB) provides interfaces to several of these libraries, but for this work, we will use its own, internal linear algebra package called Iterative Solver Template Library (ISTL) [23, 22] instead; as we focus on the problem assembly phase, the solvers contained in this library are entirely sufficient for our purpose.

1.3 Structure and Contribution

The main contribution of the present work is the creation of a software framework for the solution of multi physics and in particular multi domain problems by means of grid-based discretizations on top of DUNE and its discretization module PDELAB.

After a short introduction to partial differential equations and their solutions using general finite element type methods, we give a high level overview of the DUNE software framework that forms the basis of our implementation and describe a number of advanced C++ programming techniques required to understand a number of implementation techniques presented in later chapters.

The main body of our work is devoted to our software framework. In Chapter 3, we introduce a DUNE meta grid that adds subdomain support to any existing DUNE grid implementation, a feature which forms the foundation for our high-level multi domain simulation framework. Chapter 4 is dedicated to a general introduction into multi domain problems and a high-level overview over this simulation framework. In Chapter 5, we introduce a C++ template library for trees of function spaces and related objects that lies at the heart of our framework and the underlying PDELAB toolbox. Chapter 6 is concerned with controlling the order of entries in the assembled vectors and matrices and presents a flexible framework to generate different orders by annotating function spaces; a feature which is required for the use of advanced linear solvers typically required in multi domain simulations. In Chapter 7 we explain some internals of the implementation of our framework. Chapter 8 contains the last major part of our work, where we demonstrate the application of our implementation to a pair of model problems to illustrate its ability to speed up application development and quantify its performance overhead in relation to standard (non multi domain) DUNE. Moreover, we show its impact on real-world applications, where our software was used by other scientists to develop and perform their multi domain simulations. We describe how these projects have benefited from the infrastructure provided by our framework.

Finally, in Chapter 9 we provide a conclusion and point out a number of areas where our work could be further extended or improved.

CHAPTER 2

Fundamentals

In this chapter, we give a short overview of several topics that are fundamental to the understanding of our work. In particular, we introduce PDEs and their numerical solution using the Finite Element Method (FEM) with a focus on establishing the notation and giving an overview of the assembly process, i.e. the calculation of vectors and matrices describing the problem. After that, we continue with a presentation of the DUNE software framework that forms the foundation of our own software implementations; finally, there is a short summary of advanced C++ programming techniques which are required to understand many of our implementation decisions.

2.1 Partial Differential Equations

This section gives a very compressed overview of PDEs and their numerical solution. Is is mostly aimed at introducing the necessary terminology and notation required for later chapters and is neither exhaustive nor mathematically rigorous; for a more thorough treatise, we refer to [16], which this introduction is also largely based on.

A general linear, scalar second order PDE on the open domain $\Omega \subset \mathbb{R}^d$ can be written as

$$Lu = \nabla \cdot (\boldsymbol{D}(x)\nabla u) + \boldsymbol{b}(x) \cdot \nabla u + c(x)u = f \text{ in } \Omega \tag{2.1}$$

with the unknown $u : \Omega \to \mathbb{F}$, where $\mathbb{F}$ will normally be $\mathbb{R}$ or $\mathbb{C}$, the spatial parameter fields $\boldsymbol{D} : \Omega \to \mathbb{F}^{d\times d}$, $\boldsymbol{b} : \Omega \to \mathbb{F}^d$, $c : \Omega \to \mathbb{F}$ and the right hand side $f : \Omega \to \mathbb{F}^d$. While our software framework also supports calculations with complex numbers, we will for simplicity assume $\mathbb{F} = \mathbb{R}$ in the following.

Based on the values of $\boldsymbol{D}(x)$ and $\boldsymbol{b}(x)$, (2.1) is called

elliptic in x, if $\boldsymbol{D}(x)$ is positive or negative definite,

hyperbolic in x, if all eigenvalues of $\boldsymbol{D}(x)$ are nonzero and all but one of those eigenvalues have the same sign, or

parabolic in x, if exactly one eigenvalue is zero, all remaining eigenvalues share the same sign and the matrix $(\boldsymbol{D}(x), \boldsymbol{b}(x))$ has full rank.

A priori, this classification is a local property. In many cases, however, it holds for all $x \in \Omega$ and thus becomes a global property of the PDE in question.

The terminology of this classification is rooted in the two-dimensional setting, where, depending on the values of the matrix $\boldsymbol{D}$, the level set functions of the quadratic form $d_{11}x_1^2 + 2d_{12}x_1x_2 + d_{22}x_2^2$ take on the shape of the corresponding geometric curves.

A PDE does not necessarily fall into one of the above categories, as the classification is not exhaustive; moreover, the character of the equation can vary across Ω.

A very simple example of a stationary, scalar PDE is the stationary convection-diffusion-reaction equation. It describes the spatial distribution of a concentration u in the presence of the three processes diffusion, convection and reaction and is given by

$$\begin{aligned} -\nabla \cdot (\boldsymbol{D}\nabla u) + \boldsymbol{b} \cdot \nabla u + cu &= f \text{ in } \Omega \subset \mathbb{R}^d, \\ u &= g \text{ on } \Gamma_D, \\ -\nabla u \cdot \boldsymbol{n} &= j \text{ on } \Gamma_N, \end{aligned} \tag{2.2}$$

with a symmetric, positive definite *diffusion tensor* $\boldsymbol{D} \in \mathbb{R}^{d\times d}$, a convection *velocity* $\boldsymbol{b} \in \mathbb{R}^d$ and a *reaction rate* $c \in \mathbb{R}$. Here, $\boldsymbol{n}$ denotes the *unit outer normal vector*, g prescribes the *Dirichlet* boundary conditions and j the *Neumann* boundary conditions. Together, Γ_D and Γ_N form a partitioning of the boundary of Ω such that $\Gamma_D \cap \Gamma_N = \emptyset$ and $\partial\Omega = \Gamma_D \cup \Gamma_N$. If $g = 0$, the Dirichlet boundary conditions are called *homogeneous*.

2.1.1 Weak Form

Finding a solution u to a second order PDE like (2.2) requires that $u \in C^2(\Omega)$, i.e. that u be twice continuously differentiable everywhere in Ω. This is a very limiting restriction because it e.g. precludes approximating u by a piecewise linear function. In order to partially overcome this restriction, we smoothen the original equation by applying the variational principle, i.e. we multiply the equation by a suitable *test function* and integrate over the domain. The result is called the *weak formulation* of the PDE.

Given the convection-diffusion-reaction equation (2.2) with homogeneous Dirichlet conditions on the whole boundary, we choose a test function v that vanishes at the boundary and integrate over Ω, which yields

$$-\int_\Omega \nabla \cdot (\boldsymbol{D}\nabla u)v \,\mathrm{d}x + \int_\Omega \boldsymbol{b} \cdot \nabla uv + cuv \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{2.3}$$

Integrating the first term by parts and discarding the resulting boundary integral, which disappears due to $v = 0$ on $\partial\Omega$, we get

$$\int_\Omega (\boldsymbol{D}\nabla u) \cdot \nabla v \,\mathrm{d}x + \int_\Omega \boldsymbol{b} \cdot \nabla uv + cuv \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{2.4}$$

We then define bilinear and linear forms

$$a(u, v) = \int_\Omega (\boldsymbol{D}\nabla u) \cdot \nabla v + \boldsymbol{b} \cdot \nabla uv + cuv \,\mathrm{d}x \qquad \text{and} \qquad l(v) = \int_\Omega fv \,\mathrm{d}x.$$

Using these forms, our problem can be written in its weak form as

$$\text{Find } u \in U: \qquad a(u, v) = l(v) \quad \forall\, v \in V. \tag{2.5}$$

A solution u of (2.5) is called a *weak solution*. A weak solution does not necessarily satisfy the strong formulation (2.2); solutions of the strong formulation have to be in $C_0^2(\Omega)$, while the weak formulation only requires the first derivative. In fact, the weak solution only has to be in C^1 almost everywhere because the weak formulation is based on the notion of a weak differential that smoothens out lower-dimensional jumps in the differential.

PDELab uses a slightly modified version of (2.5): We introduce the *residual form* $r(u, v) = a(u, v) - l(v)$ and obtain the *residual formulation* of the problem:

$$\text{Find } u \in U: \qquad r(u, v) = 0 \quad \forall\, v \in V. \tag{2.6}$$

As we will see later on, this formulation has the advantage that we can naturally extend it to nonlinear problems by simply dropping the requirement that $r(u, v)$ has to be linear in u.

In general, the *ansatz space* U and the *test space* V do not have to be identical. In this very general setting, the variational approach described above is called a *Petrov-Galerkin* method. The majority of real-world applications, however, assume $U = V$, which is called a *Galerkin* method.

Both PDELab and the extension framework presented in this work support Petrov-Galerkin schemes, but in order to simplify the notation, we will restrict the following discussion to Galerkin methods with identical ansatz and test spaces, as having different ansatz and test spaces does not fundamentally change the applicability of the multi domain approaches we introduce in this thesis.

2.1.2 Existence and Uniqueness

A PDE problem is called well-posed if it has a solution, if that solution is unique and if the solution depends continuously on the data (boundary conditions and parameter functions). While there are multiple theorems that provide necessary and / or sufficient conditions for the well-posedness of a PDE problem, we restrict ourselves to stating the most well-known of those results, which gives a sufficient condition for well-posedness that is reasonably easy to verify:

Theorem 2.1 (Lax-Milgram)**.** *Let V a Hilbert space, $a \in \mathcal{L}(V \times V; \mathbb{R})$ and $l \in V'$. Then (2.5) is well-posed if a is* coercive*, i.e. if there is an $\alpha > 0$ such that*

$$a(u, u) \geq \alpha \|u\|^2 \quad \forall\, u \in V$$

and if the following a-priori estimate holds:

$$\|u\|_V \leq \frac{1}{\alpha} \|l\|_{V'} \quad \forall\, l \in V'.$$

A direct proof of this theorem can be found in [16]. For the remainder of this work, we will in general assume that any problem we are investigating satisfies Lax-Milgram or a similar well-posedness criterion and we are thus able to obtain and solve a discretized version of the problem.

2.1.3 Discrete Problem

There is only a very limited number of PDEs for which an analytical solution is known; typically, it will only be possible to calculate a numerical approximation of the solution. In general, solving a PDE equates to finding a function $\psi : \Omega \to \mathbb{R}$ which satifies (2.1) for every $x \in \Omega$. This problem cannot be directly tackled numerically, as the underlying function space V is of infinite dimension. Any numerical solution scheme thus necessarily involves a *discretization process* with the ultimate goal of finding a well-suited finite-dimensional subspace V^h that yields a good approximation of V.

If we assume that we have found a suitable finite-dimensional space V^h of dimension N (cf. Section 2.2 for how to construct such a space), we can construct a basis $\Phi = \{\varphi_i\}_{i=0,\dots,N-1}$ for V^h and represent any function $\psi \in V^h$ as a linear combination of those basis functions:

$$\psi = \sum_{i=0}^{N-1} u_i \varphi_i, \quad u_i \in \mathbb{R}. \tag{2.7}$$

The coefficients u_i are usually called degrees of freedom (DOFs) and the vector $\boldsymbol{u} = (u_0, \dots, u_{N-1})$ the DOF vector. For a fixed basis Φ, this vector is a unique representation of the function ψ that can be stored in a computer as a vector of floating point numbers; ultimately, the goal of any PDE simulation is to obtain the DOF vector $\boldsymbol{u}$ of the solution.

From a high-level point of view, the discretized version of a problem in residual form (2.5) can still be written the same way as in the continuous case:

$$\text{Find } \psi \in V^h: \qquad r^h(\psi, \varphi) = 0 \quad \forall\, \varphi \in V^h. \tag{2.8}$$

Note that the problem structure has not changed at all; the function space V and the residual form $r(\cdot,\cdot)$ have just been replaced by their discrete counterparts V^h and r^h. We then exploit the fact that $r^h(\psi, \varphi)$ is a bilinear form and it is thus sufficient to test equation (2.8) with the basis functions φ_i of V^h. Replacing ψ by (2.7), we arrive at the finite-dimensional algebraic problem

$$\text{Find } \boldsymbol{u} \in \mathbb{R}^N: \qquad r^h\Big(\sum_{j=0}^{N-1} u_j \varphi_j, \varphi_i\Big) = 0, \quad i = 0, \dots, N-1 \tag{2.9}$$

which describes the PDE problem in terms of the DOF vector $\boldsymbol{u}$. For further notation convenience, we finally introduce the discrete residual operator $\mathcal{R}^h$ as

$$\mathcal{R}^h : \mathbb{R}^N \to \mathbb{R}^N, \quad \Big(\mathcal{R}^h(\boldsymbol{u})\Big)\Big|_i = r^h\Big(\sum_{j=0}^{N-1} u_j \varphi_j, \varphi_i\Big) \tag{2.10}$$

and can then write the algebraic problem (2.9) as

$$\text{Find } \boldsymbol{u} \in \mathbb{R}^N: \qquad \mathcal{R}^h(\boldsymbol{u}) = \emptyset. \tag{2.11}$$

2.1.4 Systems of Equations

Most multi physics problems are concerned with models that describe more than a single, scalar variable. As an example, consider the Navier-Stokes equations, which describe the motion of a fluid based on its velocity $\boldsymbol{v}$ and pressure p:

$$\nabla \cdot \Big(2\mu \mathbb{D}(\boldsymbol{v}) - p\mathbb{I}\Big) + \rho \boldsymbol{v} \cdot \nabla \boldsymbol{v} = f \text{ in } \Omega, \tag{2.12a}$$

$$\nabla \cdot \boldsymbol{v} = 0 \text{ in } \Omega, \tag{2.12b}$$

$$\mathbb{T}(\boldsymbol{v}) \cdot \boldsymbol{n} = \boldsymbol{j_N} \text{ on } \Gamma_N \subset \partial\Omega. \tag{2.12c}$$

$$\tag{2.12d}$$

Here, μ denotes the viscosity and ρ the density of the fluid. f is an external force like gravity, $\mathbb{D} = \frac{1}{2}\Big(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^T\Big)$ is the symmetric deformation tensor and $\mathbb{T} = 2\mu\mathbb{D}(\boldsymbol{v}) - p\mathbb{I}$ is the stress tensor.

This problem has a more complicated structure than the scalar convection-diffusion-reaction equation:

- The function space U for this problem is vector-valued: There are d velocity components v_i as well as the pressure p, so U has $d+1$ components. Let us initially consider the velocity: we have written the equations in terms of

the overall velocity $\boldsymbol{v}$ for better readability and thus want to consider $\boldsymbol{v}$ as a single, vector-valued variable. The corresponding velocity function space $\boldsymbol{V}$ can be constructed as a Cartesian product of scalar function spaces for each velocity component:

$$\boldsymbol{V} = V_1 \times V_2 \times \cdots \times V_d, \qquad V_i \subseteq H^1(\Omega),\ i = 1, \ldots, d.$$

In order to define the overall solution space, we repeat this construction process and define $U = \boldsymbol{V} \times P$ with an additional pressure space P. This example hints at a general construction principle for multi-component function spaces: Instead of regarding them as a single, arbitrarily complex space, we can construct them by recursively combining elementary function spaces into bigger units, resulting in a structure that computer scientists will directly recognize as a *tree*. As we will see in later chapters, this construction principle is central to both PDELab and our multi domain extensions.

- Equations (2.12a) and (2.12b) give $d+1$ scalar equations for the $d+1$ scalar variables of the problem. We treat each of those equations the same way as in the scalar case by multiplying with a test function from the test space of an appropriate scalar variable and partially integrating the result, if necessary. Afterwards, we sum up the resulting equations into a single residual r for the complete system, which becomes

$$\begin{aligned} r\Big((\boldsymbol{v}, p), (\boldsymbol{w}, q)\Big) = \quad & \int_\Omega (p\mathbb{I} - 2\mu\mathbb{D}(\boldsymbol{v})) \cdot \nabla \boldsymbol{w} \,\mathrm{d}x + \int_\Omega (\rho \boldsymbol{v} \cdot \nabla \boldsymbol{v}) \cdot \boldsymbol{w} \,\mathrm{d}x \\ & - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{w} \,\mathrm{d}x + \int_{\Gamma_N} \boldsymbol{j} \cdot \boldsymbol{w} \,\mathrm{d}s + \int_\Omega (\nabla \cdot \boldsymbol{v}) q \,\mathrm{d}x \end{aligned}$$

Note that due to the convection term $\rho \boldsymbol{v} \cdot \nabla \boldsymbol{v}$, the residual form is not linear in $\boldsymbol{v}$ anymore.

Finally, we again arrive at a problem of the form (2.8):

$$\text{Find } (\boldsymbol{v}, p) \in \boldsymbol{V} \times P: \qquad r\Big((\boldsymbol{v}, p), (\boldsymbol{w}, q)\Big) = 0 \quad \forall\, (\boldsymbol{w}, q) \in \boldsymbol{V} \times P.$$

We can see that the residual formulation naturally extends to systems of equations. Moreover, it also allows for a uniform treatment of both linear and nonlinear problems (although a nonlinear problem will of course necessitate a nonlinear solver like a Newton scheme).

2.1.5 Instationary Problems

Up until now, we have only regarded time-independent equations, but the majority of real-world applications is concerned with instationary problems. As an example,

we extend the convection-diffusion-reaction equation (2.2) to an instationary model over the time interval $(t_0, t_0 + T)$:

$$\begin{aligned} \partial_t u - \nabla \cdot (\boldsymbol{D}\nabla u) + \boldsymbol{b} \cdot \nabla u + cu &= f \text{ in } \Omega \times \Sigma = (t_0, t_0 + T), \\ u &= g \text{ on } \Gamma_D, \\ -\nabla u \cdot \boldsymbol{n} &= j \text{ on } \Gamma_N, \\ u(\cdot, t_0) &= u_0 \text{ at } t = t_0. \end{aligned} \tag{2.13}$$

The main differences here are the time derivative in the actual PDE and the initial condition u_0. Instationary problems are typically solved by separating the problem into a spatial and a temporal component and then treating the temporal part as a system of ODEs for the spatial DOFs. The weak form of the problem is given by

$$\begin{aligned} \frac{d}{dt}\int_\Omega uv \,\mathrm{d}x + \int_\Omega (\boldsymbol{D}\nabla u) \cdot \nabla v + (\boldsymbol{b} \cdot \nabla u)v + cuv \,\mathrm{d}x \\ - \int_\Omega fv \,\mathrm{d}x + \int_{\Gamma_N} jv \,\mathrm{d}s = 0 \quad \forall\, v \in V, t \in \Sigma. \end{aligned}$$

We introduce a second residual form $m(u, v; t) = \int_\Omega uv \,\mathrm{d}x$ and discretize the problem in space. The problem can then be cast into a slightly extended version of our established residual formulation:

$$\text{Find } u^h(\boldsymbol{x}, t) \in V^h : \ \frac{d}{dt} m^h(u^h, v^h; t) + r^h(u^h, v^h; t) = 0 \quad \forall\, v^h \in V^h, t \in \Sigma. \tag{2.14}$$

Here, $m^h(u^h, v^h; t)$ represents the temporal part of the problem; note that r^h carries over unchanged from the stationary problem (2.6). After discretizing (2.14) in time, we can apply an ODE integrator to the problem. Taking the implicit Euler method as an example, we pick a sequence of time points $(t^0 = t_0, t^1, \ldots, t^N = t_0 + T)$ with $t^{n-1} < t^n, n = 1, \ldots, N$ and set $k^n = t^{n+1} - t^n$. Approximating the time derivative in (2.14) with backward differences then yields a sequence of fully discrete problems

$$\text{Find } \boldsymbol{u}^{n+1} \in \mathbb{R}^N : \quad \frac{\mathcal{M}(\boldsymbol{u}^{n+1}; t^{n+1}) - \mathcal{M}(\boldsymbol{u}^n; t^n)}{t^{n+1} - t^n} + \mathcal{R}(\boldsymbol{u}^{n+1}; t^{n+1}) = \emptyset,$$

where $\mathcal{R}$ is defined as in (2.10) and $\mathcal{M}$ is constructed in equivalent fashion from $m^h(u^h, v^h; t)$. We thus have to find the solution to one (non)linear algebraic problem per time step. This approach naturally extends to higher-order one step methods as well as explicit time integrators, but we limit our introduction to this simple example of an implicit Euler scheme and refer the reader to [63, 64] for a detailed overview of the numerical solution of ODEs.

Apart from the additional time dependence, $\mathcal{R}$ is identical to the residual operator (2.2) of the stationary problem, and while we now need two residual operators, $\mathcal{M}$ has the same structure as the original residual; this will be important at the implementation level, where we can use the same software infrastructure to assemble both $\mathcal{R}$ and $\mathcal{M}$.

2.1.6 Solution of the Algebraic Problem

Given an algebraic problem of the form (2.11), we can apply the following solution algorithm based on a damped Newton method: We start with an initial guess $\boldsymbol{u}^0 \in U$, which should be chosen in such a way that it fulfills any strongly imposed Dirichlet boundary conditions. We then compute $\boldsymbol{r}^0 = \mathcal{R}(\boldsymbol{u}^0)$ and set $k = 0$. Subsequently, the following steps are repeated until convergence (i.e. $\|\boldsymbol{r}_k\| < \epsilon$):

1. Assemble Jacobian matrix $\boldsymbol{A}^k = \nabla\mathcal{R}(\boldsymbol{u}^k)$.
2. Solve $\boldsymbol{A}^k\boldsymbol{z}^k = \boldsymbol{r}^k$ with some linear solver.
3. Update $\boldsymbol{u}^{k+1} = \boldsymbol{u}^k - \sigma^k\boldsymbol{z}^k$ with $\sigma \in (0, 1]$.
4. Compute new residual $\boldsymbol{r}^{k+1} = \mathcal{R}(\boldsymbol{u}^{k+1})$.
5. Set $k = k + 1$.

As we can see, we need a way to compute the residual vector $\mathcal{R}(\boldsymbol{u})$ and the Jacobian matrix $\nabla\mathcal{R}(\boldsymbol{u})$. The computation of these two quantities is called *problem assembly*; our thesis presents a software framework for performing this assembly in the case of more complicated multi domain problems as introduced in Chapter 4. Note that for a linear problem like the convection-diffusion-reaction example, the Jacobian $\nabla\mathcal{R}(\boldsymbol{u})$ is a constant; consequently, the Newton method will converge in a single iteration, making this solution procedure viable for both linear and nonlinear problems.

2.2 The Finite Element Method

Up until now, we have not made any assumptions about the nature of the finite-dimensional space V. There are a number of approaches for choosing this space, but in this work, we concentrate on Finite Element (FE) methods, which we take to mean general Petrov-Galerkin type methods, including classical continuous FE, finite volumes and Discontinuous Galerkin (DG) discretizations.

In all of these approaches, the spatial domain is discretized with a grid, and the solution is constructed as a sum of piecewise polynomial functions that each have support on a small number of grid cells. This small support causes the vast majority of terms in the Jacobian $\nabla\mathcal{R}(\boldsymbol{u})$ to vanish (because the involved test and trial basis functions do not overlap). As a consequence, the linear systems which arise from FEM discretizations tend to be extremely sparse.

2.2.1 Tessellation of the Spatial Domain

In order to construct a discrete function space, the spatial domain Ω has to be discretized. While there are other possibilities, this is usually done by creating a tessellation (also called mesh or grid).

Definition 2.1 (Tessellation)**.** *Let Ω a domain in $\mathbb{R}^d$. Then a* tessellation *is a finite set $\mathcal{T} = \{T_1, T_n\}$ of bounded domains T_i called* cells *(or elements) that form a disjoint partitioning of Ω:*

$$\overline{\Omega} = \bigcup_{i=1}^{N} \overline{T_i}, \quad T_i \cap T_j = \emptyset \; \forall \, i \neq j.$$

We define the size $h(T)$ of a cell T by $h(T) = \operatorname{diam} T = \max_{x,y \in \overline{T}} \|x - y\|$ and the mesh size *h by $h = \max_{T \in \mathcal{T}} h(T)$.*

For a one-dimensional domain $\Omega = (a, b)$ this amounts to finding suitable points

$$a = x_0 < x_1 < \cdots < x_N = b$$

and letting

$$\mathcal{T} = \{(x_{i-1}, x_i) : i = 1, \ldots, N\}.$$

In higher dimensions, meshes are typically constructed using *simplices* or *cuboids* (rectangles, hexahedra). This poses an additional challenge when Ω is not of polygonal shape. In that case, the tessellation will only be an approximation of the actual domain. Finding a good mesh can be a very challenging problem for complex geometries and heterogeneous PDEs, a thorough overview of the problem can be found in Ern and Guermond [48]; in the following, we will always assume that a suitable tessellation $\mathcal{T}$ for our problem has already been found.

2.2.2 Finite Element Spaces

Over time, different problem settings have led to the creation of a wide variety of finite element spaces. In this section, we provide two examples of well-known FE families. Please note that there are a lot of other important spaces, which guarantee different properties of the weak solution, such as Nedelec elements, which are in H_{curl} (that is, the tangential component of their gradient is continuous across element boundaries) and Raviart-Thomas elements, which can be used for solutions that have to form part of H_{div} (continuous normal component of the gradient across element boundaries). A very comprehensive overview can be found in [82].

P_k spaces are probably the best known type of continuous FE spaces, i.e. spaces which guarantee that the overall solution will be in $\mathcal{C}^0$. They require a simplicial mesh and are defined by

$$P_k(\mathcal{T}) = \{u \in C^0(\overline{\Omega}) : u|_{\overline{T}} \in \mathbb{P}_k^d \, \forall \, T \in \mathcal{T}\}, \tag{2.15}$$

where $\mathbb{P}_k^d$ denotes the space of polynomials of degree at most k in $\mathbb{R}^d$. Typically, they are represented by a Lagrange basis. Apart from being orthonormal, such a basis has the beneficial property of yielding a *nodal basis*, which makes

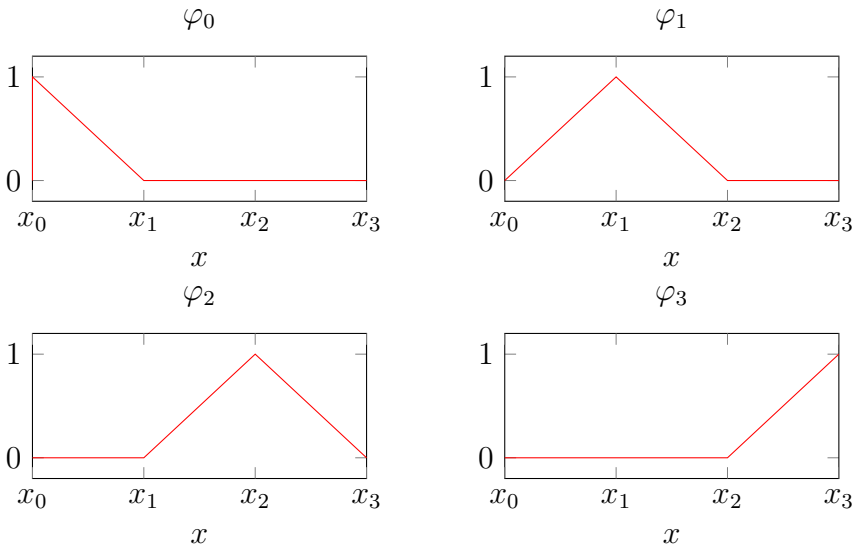


Figure 2.1 — Basis functions for a P_1 FEM space in 1D

it possible to interpolate data by simply evaluating the data at the nodes. An example of a 1D P_1 space is shown in Figure 2.1. Its construction can be extended to higher polynomial orders and dimension in a generic way (cf. [16]). According to the placement of the associated node, each basis function in a nodal basis can be associated with a unique grid entity (vertex, cell, facet, edge, etc.). In a computer implementation, the corresponding DOF is stored using this relationship.

A similar family of spaces also exists for hexahedral meshes; they are denoted by Q_k and differ slightly from P_k spaces in that their construction requires a reference element (cf. [16]).

Finite Volume spaces belong to a different class of function spaces called *broken Sobolev spaces*. Like the more "standard" FE spaces introduced above, these spaces consist of piecewise functions on the individual grid cells, but unlike those, they do not contain any built-in coupling between the per-cell functions. Figure 2.2 depicts the basis functions of such a space for a simple 1D setting. If the per-cell solution is not a constant, but a polynomial, the corresponding space is called a Discontinuous Galerkin (DG) space. Methods based on these broken spaces need to ensure global solution coherence by means of additional residual terms for the cell-cell intersections; these additional integrals penalize jumps of the solution (and / or its derivatives) to enforce the global continuity of the solution, but only do so in a weak sense. Discontinuous spaces and the corresponding methods are well suited to problems which contain discontinuous features likes shocks or discontinuities in their data.

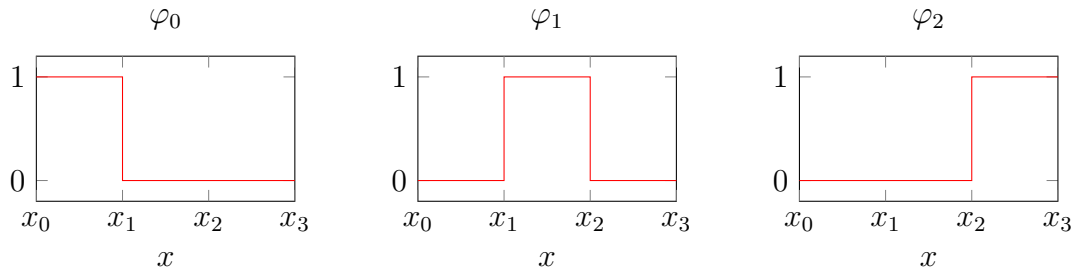


Figure 2.2 — Basis functions for a finite volume space in 1D

2.2.3 Reference Elements and Local Function Spaces

FE basis functions are typically defined in terms of a *reference element* $\hat{\Omega}_T$ for the grid cell T, for example the unit square or the unit triangle in 2D. For every grid cell T, the grid manager provides a geometry transformation $\mu_T : \hat{\Omega}_T \to \Omega_T$ that maps from the reference element to the actual geometry of the grid cell. In order to evaluate a scalar function $u^h \in U^h$ at position $\boldsymbol{x}$, we have to map the world space coordinate $\boldsymbol{x}$ into the reference space of each grid cell, where we can then evaluate the basis functions in the (cell-specific) reference space coordinate $\hat{\boldsymbol{x}}$. Taking these transformations into account, the basis representation of u^h becomes

$$u^h(\boldsymbol{x}) = \sum_{T \in \mathcal{T}} \sum_{l=0}^{n(T)-1} \boldsymbol{u}_{g(T,l)} \hat{\varphi}_{T,l}\Big(\mu_T^{-1}(\boldsymbol{x})\Big).$$

Here, $n(T)$ denotes the number of basis functions on T, $\hat{\varphi}_{T,l}$ denotes the l-th of these basis functions defined in terms of the reference element $\hat{\Omega}_T$, and $g(T,l)$ maps the local index l of this basis function to the index of the corresponding entry in the global coefficient vector $\boldsymbol{u}$.

As this example shows, interactions with a function u^h (represented by its DOF vector $\boldsymbol{u}$) and the underlying function space U^h are typically done by iterating over grid cells and performing a number of operations on the basis functions associated with each grid cell. We can simplify those cell-local operations by introducing a local function space U_T containing only those locally active basis functions and a corresponding restricted coefficient space $\boldsymbol{U}_T = \mathbb{R}^{n(T)}$. In order to map between the global and local spaces, we introduce a restriction operator R_T by

$$R_T : \boldsymbol{U} = \mathbb{R}^N \to \boldsymbol{U}_T, \quad \Big(R_T(\boldsymbol{u})\Big)_i = (\boldsymbol{u})_{g(T,i)}, \ i = 0, \dots, n(T) - 1. \tag{2.16}$$

2.2.4 Evaluation of Element-wise Residual Contributions

Considering this split into cell-local operations and grid iteration, we can rewrite the residual operator $\mathcal{R}$ (2.10) in a similar fashion to the basis representation as

$$\mathcal{R}(\boldsymbol{u}) = \sum_{T \in \mathcal{T}} R_T^T \alpha_T^{\text{vol}}(R_T \boldsymbol{u}) + \sum_{\tau \in \partial\mathcal{T}} R_\tau^T \alpha_\tau^{\text{bnd}}(R_\tau \boldsymbol{u}), \tag{2.17}$$

where the operator $\alpha_T^{\text{vol}} : \boldsymbol{U}_T \to \boldsymbol{U}_T$ and its counterpart α_τ^{bnd} encode the weak form of the actual PDE, i.e. the integrals contained in the weak form after restricting it to the current cell T or boundary intersection τ, respectively. Continuous FE methods only require the volume and boundary terms shown here; DG schemes contain additional α^{skel} terms for internal intersections between neighboring grid cells.

Looking back at the basic advection-diffusion-reaction problem, the volume integral portion of its weak form (2.4) corresponds to the following volume integral kernel, where we have assumed $\boldsymbol{b} = 0$ (i.e. no convection) for simplicity:

$$\begin{aligned}\left(\alpha_T^{\text{vol}}(u_T)\right) = \int_{\hat{\Omega}_T} \Bigg\{ \Bigg[\sum_{l=0}^{n(T)-1} (u_T)_l (\boldsymbol{D} \nabla \mu_T(\hat{\boldsymbol{x}}))^{-T} \nabla_{\hat{\boldsymbol{x}}} \hat{\varphi}_{T,l}(\hat{\boldsymbol{x}}) \Bigg] \cdot \mu_T(\hat{\boldsymbol{x}}))^{-T} \nabla_{\hat{\boldsymbol{x}}} \hat{\varphi}_{T,m}(\hat{\boldsymbol{x}}) \\ + c \Bigg[\sum_{l=0}^{n(T)-1} (u_T)_l \hat{\varphi}_{T,l}(\hat{\boldsymbol{x}}) \Bigg] \hat{\varphi}_{T,m}(\hat{\boldsymbol{x}}) - f \hat{\varphi}_{T,m}(\hat{\boldsymbol{x}}) \Bigg\} \det \nabla \mu_T(\hat{\boldsymbol{x}}) \, \mathrm{d}\hat{\boldsymbol{x}}\end{aligned}$$

Typically, this integral will be evaluated numerically using a quadrature rule for the reference element. Note that the integral kernel does not make any assumptions about the underlying FE space, apart from requiring it to be continuous. As long as the numerical integration order is chosen sufficiently high, this kernel will work for arbitrary P_k and Q_k spaces.

In order to apply the solution procedure outlined in Section 2.1.6, we also need to assemble the Jacobian matrix. This process is very similar to the residual assembly; it mainly differs in the fact that instead of the diverse α functions, it employs a set of kernels that calculate local matrix contributions. For a detailed overview, we refer to [101].

2.2.5 Finite Element Assembly

As shown above, the discretized weak form of a PDE consists of integrals over grid cells and intersections of cells with other cells / the domain border. In order to numerically calculate this residual (often called *form assembly*), we have to devise an algorithm for (2.17). This process is a central part of any FEM framework; consequently, there is a large number of implementations with different features and based on different variants of the weak formulation [10, 11, 12, 68, 115]. As our work is based on the DUNE-PDELAB framework, we quickly restate its version of the algorithm as described in [101].

Algorithm 2.1 gives a high-level overview of the assembly process in DUNE-PDELAB: It directly corresponds to equation (2.17) and is realized as a nested iteration over the grid cells (line 3) and – at the second level – the intersections associated with each grid cell (line 9). It only differs in that it further splits the integration kernels into a part $\alpha^{\cdot}$ that depends on the solution $\boldsymbol{u}$ and a constant part $\lambda^{\cdot}$, which is an optimization detail that allows us to avoid the construction of the restricted

Algorithm 2.1 — Finite Element Residual Assembly. Problem-specific (user-supplied) building blocks are highlighted in **bold**.

---

function RESIDUAL(u)
  $r \leftarrow 0$
  for $T \leftarrow$ CELLS($\mathcal{T}$) **do**
    $u_l \leftarrow \mathcal{R}_T(u)$ ▷ restrict solution
    $\{\varphi_{l,i}\} \leftarrow \mathcal{R}_T(\{\varphi_j\})$ ▷ restrict trial space
    $\{\psi_{l,i}\} \leftarrow \mathcal{R}_T(\{\psi_j\})$ ▷ restrict test space
    $r_l \leftarrow \boldsymbol{\alpha_{V,T}(u_l, \{\varphi_{l,i}\}, \{\psi_{l,i}\})}$
    $r_l \leftarrow r_l + \boldsymbol{\lambda_{V,T}(\{\psi_{l,i}\})}$
    for $I \leftarrow$ INTERSECTIONS(T) **do**
      if $I \subset \partial\overline{\Omega_h}$ **then** ▷ boundary intersection
        $r_l \leftarrow r_l + \boldsymbol{\alpha_{B,I}(u_l, \{\varphi_{l,i}\}, \{\psi_{l,i}\})}$
        $r_l \leftarrow r_l + \boldsymbol{\lambda_{B,I}(\{\psi_{l,i}\})}$
      else ▷ interior intersection
        $T_n \leftarrow$ NEIGHBOR(T, I)
        $u_n \leftarrow \mathcal{R}_{T_n}(u)$ ▷ restrict solution to neighbor
        $\{\varphi_{n,i}\} \leftarrow \mathcal{R}_{T_n}(\{\varphi_j\})$ ▷ restrict trial space
        $\{\psi_{n,i}\} \leftarrow \mathcal{R}_{T_n}(\{\psi_j\})$ ▷ restrict test space
        $r_l \leftarrow r_l + \boldsymbol{\alpha_{S,I}(u_s, \{\varphi_{l,i}\}, \{\psi_{l,i}\}, u_n, \{\varphi_{n,i}\}, \{\psi_{n,i}\})}$
        $r_l \leftarrow r_l + \boldsymbol{\lambda_{S,I}(\{\psi_{l,i}\}, \{\psi_{n,i}\})}$
    $r' \leftarrow \mathcal{R}_T^- 1(r_l)$ ▷ prolongate residual update
    $r \leftarrow r + r'$ ▷ accumulate residual contributions
  return r

---

solution u_T in some cases and to speed up the calculation of the Jacobian matrix by numerical differentiation, where we can skip over the constant terms.

The majority of the contributions in our thesis are directly motivated by some of the operations in this algorithm and by the manipulated objects. The layout of the algorithm already shows that only the element- and intersection-local integrals (the various functions designated α and λ) are entirely problem-dependent. The remainder of the algorithm always stays the same: it performs a number of operations that are only described in very general terms like "restrict trial space". Likewise, the exact nature of objects like function spaces and the restriction / prolongation operators is not specified in the algorithm. PDELAB contains a very generic implementation of this algorithm, and we have extended this implementation

to cope with the specific challenges of multi domain simulations:

- Most of the multi physics and multi domain problems that we want to simulate are systems of PDEs and the corresponding function spaces are thus product spaces. In order to provide a mechanism to recursively compose such function spaces, we introduce a new library for trees of unrelated types in Chapter 5.
- Complex, structured problems require a great deal of flexibility in the layout of the global index set $\mathcal{I}_{U^h}$. In practice, this layout is determined by the local-to-global map g. In Chapter 6, we present a framework that constructs such maps from the tree structure of the underlying function trees and supports very flexible ordering of the entries as well as support for block-structured containers via multi indices.
- In case of multi domain problems, the algorithm is more complicated than shown here; different integrals have to be computed on different parts of the spatial domain. In Chapter 3, we introduce a mechanism that makes it possible to carve up existing grids into subdomains to describe those parts. Finally, Chapter 4 describes a software framework based on this mechanism that largely automates the process of mapping both function spaces and subproblem-specific terms of the residual form to those subdomains.

2.3 The DUNE Framework

The software implementations presented in this thesis are based on the Distributed and Unified Numerics Environment (DUNE) [18, 19], a major open source software framework for the development of grid-based simulations written in C++. It leverages a number of advanced C++ programming techniques and is based around the following design concepts:

Flexibility DUNE is based around a number of detailed interface specifications; any program built on top of these interfaces can easily exchange the underlying implementation without changes to the user code. The most important example of this is the grid interface, which allows users to switch e.g. from structured to unstructured grids or from 2D to 3D calculations by simply changing a small number of `typedef`s.

Extensibility A DUNE project is made up of separate *modules*. These modules may provide libraries and / or executable programs; a module can depend on any number of other modules. By clearly separating distinct parts of the framework (infrastructure, grid interface, linear algebra,...) into different modules, it becomes much easier to swap out certain functionality and try different simulation approaches.

Efficiency Numerical simulations require enormous amounts of computational power when applied to real-world problems. Consequently, software performance is of paramount importance, a goal that often requires highly specialized implementations. DUNE manages to achieve high efficiency despite its focus flexibility.

In order to achieve these goals, DUNE relies on the template feature of C++, which resolves all of the interface choices (grid implementations, function spaces etc.) at compile time; combined with extensive function inlining, this allows the C++ compiler to generate optimal code for the set of parameters specified by the user.

2.3.1 Components

DUNE modules fall into one of two basic categories: *core modules*, which provide the basic functionality for creating grid-based simulations and which are (mostly) essential for the creation of a DUNE-based program, and add-on modules that provide additional functionality (mostly at higher levels of abstraction).

The core modules are subject to a much more stringent development process, curated by a team of *core developers*. These modules have a common, coordinated release schedule (the current aim is for one major release per year) and provide a certain amount of API stability (for example, current functionality is normally only modified or removed after having been deprecated for at least one major release cycle). Currently, there are 5 core modules:

dune-common contains basic infrastructure used by all DUNE modules, such as the build system, dense vectors and matrices as well as support for MPI-based parallel computations.

dune-geometry provides support for the geometric primitives and operations required for FEM simulations. This includes reference elements, geometry transformations and quadrature rules and a support infrastructure for the on-the-fly creation of refinements of reference elements.

dune-grid contains the abstract grid interface API along with a self-contained parallel structured grid implementation and adapters for a number of pre-existing grid managers and additional infrastructure components that provide grid input / output (I/O) (grid factories, file parsers, Visualization ToolKit (VTK) output) and provide additional functionality built on top of the basic API. Finally, this module contains a meta grid (explained in the following section) that can be used to modify the geometry of an existing DUNE grid.

dune-istl, the *iterative solver template library*, is a collection of vectors, matrices, algorithms and solvers for sparse linear algebra.

dune-localfunctions defines an API for finite elements (interpolation, evaluation, DOF mapping) and contains implementations of many common elements, such as P_k, Q_k, DG, Brezzi-Douglas-Marini, Rannacher-Turek, Raviart-Thomas, Whitney and monomial basis functions (orthonormal and non-orthonormal).

While the core modules provide all necessary components to build a FEM simulation, doing so still requires the user to write a lot of infrastructure for tasks like function space and DOF handling, constraints processing, matrix / residual assembly and problem solution (parallel and non-linear solvers, time stepping, etc.). There are at the moment three distinct higher-level frameworks which provide this functionality (at least in part) and which are currently in development by different members of the DUNE core team:

dune-fem [34] provides abstractions for the creation of FEM simulations (e.g. function spaces, functions, linear operators). Its implementation is mostly focused on supporting local adaptivity and load-balanced parallel computations on unstructured meshes. It is available from [41].

dune-fufem, according to its maintainers (Gräser, Sack, and Sander [57] and private communication), is mostly focused on ease of use instead of performance (as exemplified by its extensive use of dynamic polymorphism, largely absent in most other DUNE modules). It is not widely available, though, stemming from an internal development effort at the Freie Universität Berlin.

PDELab [15] is a highly abstracted software framework for solving PDE-based problems that can be cast into a residual formulation. It provides a large number of spatial and temporal discretizations, function spaces with a powerful composition mechanism for problems with multiple variables, solver backends for linear and non-linear problems that can be used on overlapping as well as non-overlapping parallel grid decompositions and time integrators for explicit and implicit time-stepping methods. It is licensed under GPL2 with a run-time exception and LGPL3; see [102] for further information. PDELab has successfully been used to implement a wide variety of FE-type discretizations, including continuous FEM, Finite Volume Method (FVM), Finite Difference Method (FDM) and DG schemes.

There is also a growing number of additional grid implementations, most of which are available as open-source DUNE modules. Examples include a 2D in n-D grid [42], a cornerpoint grid [109] as well as an alternative structured grid [98, 43] and a growing number of meta grids, which are described in the next section.

Like the core modules, all of the extension modules mentioned above are freely available under open-source licenses.

2.3.2 Grid Interface

One of the most important aspects of DUNE is its very detailed mathematical definition of a grid and its programmatic interface. In the following, we introduce a number of key elements of that interface which are important in the following chapters; a complete overview can be found in [18, 19].

We consider a grid $\mathcal{T}$ as a container of *entities*: cells, vertices, edges etc. These entities can be grouped in several ways; in the following, we will use the criteria codimension and reference element. The *codimension* $c_\tau = d_\mathcal{T} - d_\tau$ of a grid entity $\tau^{()}$ is a convenient way to identify the function of an entity in a dimension-independent fashion: for example, an entity of codimension 0 is always a grid cell, independent of the dimension of the mesh. The codimension is an important tool for creating dimension-independent code. The reference element (which describes the geometric shape of an entity) is a finer criterion, as there can be multiple reference element for a given dimension, e.g. in 2D, where cells ($c = 0$) can be further subdivided into triangles and quadrilaterals. This classification is e.g. important when working with a hybrid grid, i.e. a grid that contains cells of different shape: A function space must return a different finite element for a grid cell T depending on whether T is a triangle or a quadrilateral,

We assume that a grid provides separate entity sets E_c for each codimension c. Moreover, let E_{RE} the set of all entities with reference element RE. Then the grid provides a bijective map $\mathcal{I}_\mathcal{T}^{\mathrm{RE}} : E_{\mathrm{RE}} \to \{0, \dots, |E_{\mathrm{RE}}| - 1\}$, which we call an *index map*. Evaluating this index map in forward direction is very cheap and provides the primary means for attaching data to grid entities; typically, an algorithm will place data in contiguous arrays of size $|E_{\mathrm{RE}}|$ and look up the correct offset into the array by means of $\mathcal{I}_\mathcal{T}^{\mathrm{RE}}$.

Entities in a grid can be accessed by means of an iterator interface that allows iterating over all entities of a given codimension. In addition, it is also possible to iterate over all intersections of a single mesh cell with its neighboring cells and / or the outer boundary of the mesh. As DUNE grids support h-adaptivity with hanging nodes, these intersections do not necessarily coincide with a grid edge (i.e. an entity of codimension 1).

Meta Grids

The grid interface is sufficiently fine-grained to allow for an implementation of a new grid on top of an existing DUNE grid implementation, creating a so-called *meta grid*. Meta grids are a powerful mechanism for enhancing or modifying the functionality of an existing grid, instantly providing new functionality to a wide range of different grid managers without requiring an understanding of their implementations, as any delegation happens through the common grid interface. As part of this thesis, we have developed a meta grid that adds support for subdomains to existing DUNE grid implementations (cf. Chapter 3).

2.3.3 PDELab

PDELab [102, 101] provides a number of high-level abstractions to build PDE solvers on top of DUNE grids. The multi domain simulation software developed as part of this work is realized as a set of extensions to the PDELab framework and consequently reuses many of those abstractions as well as the associated terminology. For that reason, we provide a short overview of the typical building blocks of a PDELab simulation as well as their API.

Remark 2.1. DUNE places all of its implementation into the namespace `Dune::`, and PDELab-specific functionality can be found in the namespace `Dune::PDELab::`. In order to improve the readability of the code examples in this section, we omit both of those namespace prefixes and assume that they have been imported by means of appropriate **`using`** directives.

One of the most fundamental PDELab concepts is the *function space*, which is defined in terms of a `GridView` (a read-only view of a DUNE grid), a `FiniteElementMap` that associates a specific finite element with each grid cell, a constraints engine (that is used to assemble information about constrainted DOFs) and a vector backend tag that is used to select and parameterize the vector implementation used for the DOF vectors. In general, a function space is created like this:

```
// Define type of function space
typedef GridFunctionSpace<
  LeafGridView,     // GridView on which the function space is defined
  FiniteElementMap, // map from cells to finite elements from dune-localfunctions
  Constraints,      // constraints engine (Dirichlet, hanging nodes, parallel...)
  VectorBackend     // control DOF vector structure (blocking etc.)
  > GFS;
// Instantiate fully specified function space type
GFS gfs(
  grid_view,
  finite_element_map,
  constraints_engine
  );
```

Note that due to the statically typed and templated nature of the library, PDELab components are typically created by a combination of a **`typedef`** to fill in the template parameters of the component template, followed by an instantiation of the newly defined type.

Constrained spaces do not directly store their constraints, but use an external *constraints container* for this purpose, which has to be manually created and filled with the actual constraints. For the common case of a continuous Galerkin space with Dirichlet constraints, this is achieved by

```
typedef typename GFS::template ConstraintsContainer<double>::Type C;
C cg;
constraints(constraints_parameters,gfs,cg);
```

The `constraints_parameters` can carry additional information required for the constraints assembly; here they are used to determine the boundary type (Dirichlet vs. Neumann) at a given location.

The problem-specific code written by the user is mostly concerned with the implementation of the cell- and intersection-restricted residual and jacobian terms. These are encapsulated in a class that has to conform to the `LocalOperator` concept. As this part of the simulation is a direct translation of the residual form, it will typically be provided by the simulation developer, but PDELab also includes a number of implementations for common problems, e.g. an operator for the Poisson equation:

```
typedef Poisson<
  SourceTerm,
  ConstraintsParameters,
  NeumannTerm
  > LocalOperator;
LocalOperator local_operator(
  source_term,
  constraints_parameters,
  neumann_term,
  quadrature_order
  );
```

In keeping with the largely dimension-agnostic nature of DUNE, most local operators will work for any grid dimension, making it easy to e.g. do method development with a simple 2D example and then switch to a full 3D application at a later point. They typically also do not require a specific function space; the Poisson operator shown above will work for any type of continuous function space and only requires the user to specify a sufficiently high quadrature order. The local operator contains a number of local kernels that are used by PDELab to set up the sparse matrix pattern and assemble the residual or its Jacobian (cf. Section 2.2.5). These kernels are implemented as callback methods of the `LocalOperator`, e.g. for the α^{vol} term:

```
// tell PDELab to call the alpha_volume() kernel
static const bool doAlphaVolume = true;

template<
  typename Cell,           // grid cell (for geometry information etc.)
  typename LFSU,           // local ansatz space for current cell
  typename LFSV,           // local test space for current cell
  typename X, typename R>  // local solution and residual vectors
void alpha_volume(
  const Cell& cell,
  const LFSU& lfsu, const X& x, const LFSV& lfsv,
  R& r
  ) const;
```

Note the additional flag that must be set; it is evaluated at compile time by PDELab and makes it possible to optimize the global assembly (Algorithm 2.1)

by tailoring it to the set of active kernels. If the user does not want to provide an explicit implementation of the Jacobian, PDELab can calculate it automatically by performing a numerical differentiation of the residual.

The user-provided `LocalOperator` only contains an implementation of the local residual contributions; as explained in Section 2.2.5, the remaining parts of the assembly algorithm are of a mostly problem-indepedent nature and can be provided by the framework in a generic manner. In PDELab, this job is performed by the *grid operator*, which applies the `LocalOperator` to a pair of (possibly constrained) ansatz and test function spaces. Moreover, it also handles the creation of matrices for the Jacobian, which are usually stored in a sparse format. The sparsity pattern depends on the mesh topology and the discretization scheme; matrix creation thus requires a mesh traversal to extract this information and set up the pattern. The exact format of the matrix (block structure etc.) can again be chosen by a `GridOperator` parameter. Resuming our example from above, we can define the grid operator by

```
typedef GridOperator<
  GFS,GFS,              // ansatz and test function spaces
  LocalOperator,        // user-provided local integration kernels
  MatrixBackend,        // descriptor for matrix structure / implementation
  double,double,double, // numeric types for solution, residual and jacobian
  C,C                   // constraints containers for ansatz and test spaces
  > GO;
GO grid_operator(
  gfs,cg,
  gfs,cg,
  local_operator,
  matrix_backend
  );
```

Using the grid operator, we can now easily create vectors and matrices and calculate residuals and jacobians to feed into the algebraic solver:

```
typedef typename GO::Traits::Domain Vector;
Vector solution(gfs,0.0);
Vector residual(gfs,0.0);

typedef typename GO::Traits::Jacobian Matrix;
Matrix jacobian(grid_operator,0.0);

grid_operator.residual(solution,residual);
grid_operator.jacobian(solution,jacobian);
```

PDELab also contains components to solve the assembled algebraic problem (sequential and parallel linear solvers, preconditioners, a Newton solver) as well as time integrators for implicit and explicit methods and tools for exporting the solution to a VTK file, but since our focus is on problem assembly, we omit a description of that functionality at this point.

2.4 Advanced C++ Programming Techniques

The DUNE libraries are written in C++ [72, 73]; they rely heavily on the template mechanism of the language to create abstract components and algorithms that can easily be combined in many different ways.

2.4.1 Template Programming

Template programming is much more powerful in this regard than traditional object-oriented polymorphism: Templates allow users to combine arbitrary objects without imposing any kind of language-level relationships between them; instead, objects are only required to conform to a specific set of requirements (often called *concepts* in C++) to be usable with a templated component or algorithm. A good example for this behavior are the containers that are part of the C++ Standard Template Library (STL): Assume that we have a custom class `GridFunctionSpace` representing a discrete function space and want to use a `std::vector` to store a collection of those spaces. In order to do so, we *instantiate* a special version of the `std::vector` template for our `GridFunctionSpace` type:

```
1 typedef std::vector<GridFunctionSpace> GFSVector;
2 GFSVector gfs_vector;
3 gfs_vector.push_back(gfs);
```

This causes the compiler to generate a special version of `std::vector` that is tailored to only store `GridFunctionSpace` objects and nothing else. In the same manner, we can create custom versions of the container that store `int`s, `double`s or objects of any other built-in or user-defined type. At the same time, `std::vector` imposes a number of requirements on the type of the contained objects. For example, the last operation in the code snippet above adds a copy of `gfs` to the end of the vector, an operation that requires `gfs` to be copyable. The C++ standard lists the requirements on the stored type for all of the `std::vector` operations in [C++/23.2.1/4] and [C++/23.2.3/3], which highlights the major drawback of this approach: The contract between a template class and its arguments is entirely implicit; the requirements on the arguments cannot be expressed as part of the code, but have to be documented elsewhere. If you try to combine a template with an incompatible argument, the compiler will usually generate an error with a message that seems entirely unrelated to the actual problem (e.g. a missing method definition on the argument class). Due to this lack of proper tooling, template programming exhibits a very steep learning curve and requires intimate knowledge of the implicit contracts between the individual components, which causes most programmers to avoid heavy template usage in their programs. However, in the context of scientific software the technique has seen widespread adoption in recent years, mostly because it makes a very large amount of type information available to the compiler, allowing it to perform aggressive optimization and produce code that rivals hand-tuned FORTRAN implementations in speed while being vastly

easier to maintain. In Chapter 5, we have quantified the effect of those compiler optimizations, in particular function inlining, on the performance of PDELab.

Template Specialization

In certain situations, a template could in theory work with a given set of template parameters, but the default version of the template implementation is not compatible with the given template parameter values or leads to suboptimal code. In these situations, C++ allows the programmer to provide a different implementation tailored to a specific value of the template parameter(s). This happens via a mechanism called *template specialization*; most C++ programmers have been exposed to it via the specialized implementation of `std::vector<bool>`: A single boolean variable takes up 1 byte (i.e. 8 bits) of memory, making the default vector implementation very wasteful for large collections of booleans. For that reason, the C++ standard requires a specialization `std::vector<bool>` that compresses the boolean data and use only a single bit for each stored boolean.

Template specialization becomes more complicated if the argument for which we want to specialize is a template itself; in this case, we have to resort to *partial specialization*:

```
1 // container for vector-like objects
2 template<typename Vector>
3 struct Container;
4
5 // specialization for all std::vector instantiations
6 template<typename T, typename Alloc>
7 struct Container<std::vector<T,Alloc> >;
```

Tag Dispatching

While partial specialization is a valuable tool in writing generic C++ code, the example above highlights a major drawback of the technique: Our partial specialization must provide values for *all* of `std::vector`'s template arguments. On the other hand, many programmers are not even aware of its second argument, an allocator that can be used to control how the vector allocates memory. This requirement for exactly matching the template argument signature becomes a major maintenance burden in large C++ projects with many template classes: whenever the template argument list of a class changes, we have to update all partial specializations for that template class; forgetting a single specialization will result in a compilation failure.

Tag dispatching is a mechanism to reduce this effort. It works by associating a template class with a simple *tag type* that identifies the template itself, not any specific instantiation. As an example, consider the grid function spaces in PDELab:

```
1 struct LeafGridFunctionSpaceTag {};
2
```

```
template<...>
class GridFunctionSpace
{
  ...
  typedef LeafGridFunctionSpaceTag Tag;
  ...
};

struct CompositeGridFunctionSpaceTag {};

template<...>
class CompositeGridFunctionSpace
{
  ...
  typedef CompositeGridFunctionSpaceTag Tag;
  ...
};
```

With these tags in place, algorithms and data structures can now be specialized using the tags instead of the actual template classes and thus do not have to worry about their template arguments anymore. In order to use the tag, it must appear as a template argument; for that reason, templates that rely on tag dispatch usually work in two stages: In a first step, the tag is extracted and promoted to a template argument. For example, if we want to write a `VTKExporter` class for writing a grid function to a VTK file, we can embed this step into the non-specialized declaration of that class:

```
template<typename GFS, typename Tag = typename GFS::Tag>
class VTKExporter;
```

The specializations for the different types of function spaces are then only specialized on the fixed tag type and do not make any assumptions about the template arguments of the actual function space object, greatly reducing the coupling between those components:

```
template<typename GFS>
class VTKExporter<GFS,LeafGridFunctionSpaceTag>
{
  // implementation for all instantiations of GridFunctionSpace
};

template<typename GFS>
class VTKExporter<GFS,CompositeGridFunctionSpaceTag>
{
  // implementation for all instantiations of CompositeGridFunctionSpace
};
```

2.4.2 Template Meta Programming

C++ Template Meta Programs (TMPs) exploit the type system and in particular the template mechanism to generate programs that are executed at compile time

Listing 2.1 — Simple template meta program to calculate the factorial

```
1  template<int n>
2  struct factorial
3  {
4    static const int value = n * factorial<n-1>::value;
5  };
6
7  template<>
8  struct factorial<0>
9  {
10   static const int value = 1;
11 };
```

rather than at run time. This technique was discovered during a meeting of the C++ standardization committee [118] and fully developed in [120]. Listing 2.1 shows an example of a very simple TMP that calculates the factorial of an integer number.

A TMP is invoked by instantiating the template; the template arguments then take on the role of function arguments in standard (run time) algorithms. Template meta programming differs substantially from standard programming in imperative languages like C or "normal" C++:

- First and foremost, as already stated, the program is *executed* by the compiler. This is a very important feature in the context of optimization in that it allows us to move part of the computational effort from the execution to the compilation phase; this is particularly efficient if the program is run multiple times or performs a calculation many times during a single execution.
- The only types of values available to a TMP are *C++ types* and *integral constants*, as template arguments must fall into one of those classes.
- Template meta programming falls into the category of functional programs; variables are immutable (neither types nor integral constants can be modified) and programs consist of a series of simple expressions that cannot have any side effects.

Throughout our work, we heavily rely on TMPs to create abstract, generic algorithms that operate on complex, composite objects (e.g. trees of function spaces) of unknown type; in contrast to dynamic polymorphism, the compiler can resolve all code paths at compile time and can thus turn the generic algorithm into highly optimized code. This code typically runs almost as fast as traditional C or FORTRAN code that has been hand-optimized for a specific set of data structures.

2.4.3 Improved Language Support in C++11

Template meta programming in C++ is not a feature that was designed to be part of the language; as already established above, the technique was discovered by chance in 1995 by Unruh [118]. Nevertheless, its power was quickly recognized by the community and it became the subject of intensive research [120, 119, 3] that went so far as to prove that the template mechanism in C++ forms a Turing-complete language – executed within the compiler.

Despite the success of template meta programming, the technique clearly suffers from the fact that it "abuses" C++ to do things the language was not designed for. This lack of language-level support has created idiosyncrasies like the fact that template meta functions are actually structs that "return" their value in a nested type or having to resort to obscure language properties like Substitution Failure Is Not An Error (SFINAE), which is a more powerful (but also harder to understand) alternative to template specialization. Taken together, these idiosyncrasies make TMP code very difficult to understand for programmers without experience in the field. Moreover, there are important gaps in functionality. The most obvious of those hard restrictions are probably the limitation of arithmetic to integer types and the requirement that a template argument list must be of fixed length.

As template meta programming was slowly starting to get recognized as one of the strengths of C++, the International Standards Organization (ISO) standardization committee, during its work on the C++11 revision of the language, added a number of new language features that simplify the writing of template meta programs. In the following, we highlight some of those features that are of particular importance in the scope of this work.

Variadic Templates

One of the major challenges when implementing containers for a variable number of elements of different type (e.g. a tuple) in C++03 is the lack of templates with a variable number of arguments. To work around this limitation, implementations have to choose a fixed upper bound to the number of arguments the data structure can accept and design the interface with this maximum number of arguments; shorter variants can then be realized by defaulting the template arguments to a special marker type signaling an empty argument and specializing the data structure accordingly:

```
1 // marker for empty argument slots
2 struct empty {};
3
4 // standard tuple implementation for full number of arguments
5 template<typename T1 = empty, typename T2 = empty>
6 struct tuple { /* implementation */ };
7
8 // one-argument tuple
9 template<typename T1>
```

```
10 struct tuple<T1,empty> { /* implementation */ };
11
12 // zero-argument tuple
13 template<>
14 struct tuple<empty,empty> { /* implementation */ };
```

While this pattern does work, it has major drawbacks in that it leads to massive code repetition (remember that the actual bodies of the individual variants all need to be fully spelled out), and care has to be taken to apply changes and bug fixes to all specializations. As pointed out by Gregor and Järvi [59], the resulting code is hard to maintain, causing developers to resort to *preprocessor meta programming* [77].

By contrast, variadic templates allow an arbitrary number of template arguments by adding template argument packs to the language. With these packs, our tuple type becomes

```
1 template<typename... T>
2 struct tuple { /* implementation */ };
```

and we do not require any specializations anymore. Moreover, all the unused template parameters in the original version create very long type signatures which in turn cause severe performance issues for compilers. In the example presented in [77] (the `tuple` library from TR1), compile times increase exponentially with the maximum number of arguments permitted, which the authors contrast with a reimplementation based on variadic templates that avoids those limits completely while outperforming the original version even if the latter was limited to a maximum of three arguments.

Variadic template are still a rather advanced tool in the C++ toolbox; the variadic template argument packs cannot be accessed directly, but have to be unpacked in algorithms that recursively peel off individual elements of the pack by means of partial specialization or function overloading. For example, a function that takes an arbitrary number of arguments and prints each argument to the console could be implemented like this:

```
1  // recursion terminator, matches an empty argument list
2  template<typename... T>
3  void print(const T&... t)
4  {}
5
6  // extract the first element from the pack
7  template<typename T1, typename... T>
8  void print(const T1& t1, const T&... t)
9  {
10   std::cout << t1 << std::endl;
11   // recurse into tail of the pack
12   print(t...);
13 }
```

While this programming style takes some getting used to, variadic templates have been an invaluable tool for the implementation of the more advanced features of the TypeTree library.

Type Deduction With auto and decltype

C++03 lacks a mechanism to capture the return type of a function or functor, which complicates the creation of type-agnostic, stackable functors, which are of major importance in the design of expression templates and similar constructs. To work around this problem, a library-level protocol was devised to obtain this information:

```
template<typename Expr>
struct negate {
  template<typename T>
  struct result_of {
    typedef typename Expr::template result_of<T>::type type;
  }

  template<typename T>
  typename result_of<T>::type operator()(const T& t) const {
    return - expr(t);
  }

  Expr epr;
};
```

A user of the `negate` functor will then have to invoke the `result_of` meta function to determine its return type, which it needs to know in order to store the return value. While this protocol works and has been used in major frameworks over the years (e.g. Boost MPL[61]), it is exceedingly fragile, as programmers have to take care not to forget the `result_of` meta function, which can be problematic when integrating 3rd-party code. C++11 greatly simplifies the creation of this type of generic functor with the help of the new **`decltype`** keyword and a new syntax for function declarations. Using these features (and rvalue references to ensure perfect forwarding, another deficiency of the original implementation), the above example becomes

```
template<typename Expr>
struct negate {
  template<typename T>
  auto operator()(T&& t)
  -> decltype(expr(std::forward<T>(t))) const {
    return - expr(std::forward<T>(t));
  }

  Expr epr;
};
```

At a more mundane level, the new **auto** keyword allows programmers to omit the type of a newly declared variable if its type can be deduced from the initializer expressions. Consider the following example:

```
auto i = vec,size(); // i becomes a std::size_t
auto f = std::exp(2.2); // f becomes a double
```

In general, **auto** makes the vast majority of explicit type declarations in user code redundant, greatly improving code readability, especially if those types have to be extracted from `typedefs` nested inside other variables.

While **auto** can be used to automatically deduce the type of a newly declared variable, its counterpart **decltype** makes it possible to store a deduced type in a **typedef** or use it as a template parameter, a feature that was used in the example above and that forms the basis of a new type of static polymorphism for template meta functions introduced in Chapter 5.

CHAPTER 3

Conforming Subdomains for DUNE Grids

In order to simulate multi domain problems like those mentioned in Chapter 1 (FSI, Stokes–Darcy coupling etc.), we have to manage separate meshes for those spatial subdomains within our application. We also need an efficient method for calculating the overlaps and intersections between those subdomains (as we will later see, the individual subproblems within our overall multi physics model couple via integrals over those intersections). There are two fundamental approaches to this problem:

- We can discretize each subdomain individually. This way, we are able to tailor each mesh optimally to the underlying problem. Moreover, it becomes possible to reuse existing software packages for the subproblems, even if those packages use incompatible data formats. On the other hand, this approach introduces a lot of additional complexity in order to calculate the intersections between those unrelated meshes and to transfer spatial data (e.g. solutions) between the subdomains. Despite those challenges, this architecture is used by all established software frameworks in this area, e.g. [20, 45, 47, 89], mostly due to the offered flexibility and being able to integrate existing software with a mostly black-box approach.
- Alternatively, we can start with a single, global mesh for the entire (multi domain) simulation and then designate subdomains by marking the applicable cells in that overall mesh. This mostly inverts the advantages and disadvantages of the other strategy: All parts of the simulation have to be based on a common mesh data structure, and in areas where subproblems overlap, we cannot pick individual tessellations that are optimal for each of those

subproblems. The major upsides of this approach are vastly reduced software complexity (in terms of the coupling of subproblems) and the possibility to investigate both loosely coupled and monolithic solvers at the algebra level.

With this thesis, we aim at simplifying the development and investigation of new numerical solution schemes for a wide variety of problem classes. For that reason, we have chosen the second strategy and have created the subdomain-aware grid manager `MultiDomainGrid` that forms the basis of our multi domain framework.

In this chapter, we describe the functionality of this grid manager (realized as a DUNE module) and outline its implementation. In particular, we highlight its performance characteristics and present a way of tailoring the grid to specific problems by means of a modular backend engine. The information in this chapter is based on and expands our previous work in [96].

Remark 3.1. The C++ classes that implement the grid manager introduced in this chapter are located in the namespace `Dune::mdgrid`. In order to improve the readability of the code snippets in this chapter, we will omit this namespace and assume that the two namespace `Dune` and `Dune::mdgrid` have been imported with appropriate `using` declarations.

3.1 Introduction

`MultiDomainGrid` has been developed as an add-on module for Dune-Grid and can be found in the DUNE module Dune-Multidomaingrid. It is free software and available under the same licence as the DUNE core modules (the GNU General Public Licence with a special runtime exception, for further details see [40]). In addition to Dune-Grid and its dependencies, an installation of the Boost C++ libraries [114] is also required. In particular, the code uses the Boost packages MTL and Fusion.

The following description is based on version 2.3.1 of the library, which can be downloaded from [93] or directly from the source code repository at [94]. It requires the corresponding 2.3.1 release of the DUNE core modules.

3.1.1 Overview

The functionality and basic design principle of Dune-Multidomaingrid are shown in Figure 3.1. The module is implemented in terms of two cooperating meta grids, `MultiDomainGrid` and `SubDomainGrid`: The `MultiDomainGrid` wraps an existing *host grid* and extends it with an interface for setting up and accessing subdomains. It also stores all data required to manage the subdomains. The individual subdomains are exposed as `SubDomainGrid` instances, which are very lightweight objects that combine the information from the host grid and the associated `MultiDomainGrid`. They present a subdomain as a regular DUNE grid. In

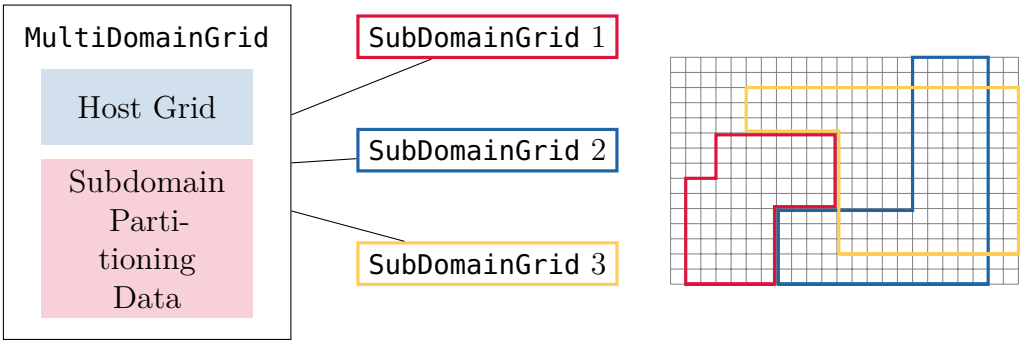


Figure 3.1 — Functionality and basic design of DUNE-MULTIDOMAINGRID: An existing host grid is wrapped and extended to support multiple subdomains, each available to the user as a distinct `SubDomainGrid`.

order to differentiate between the different subdomains, they are assigned numbers from the set $[0, N-1]$, where N denotes the maximum number of subdomains supported by the grid.

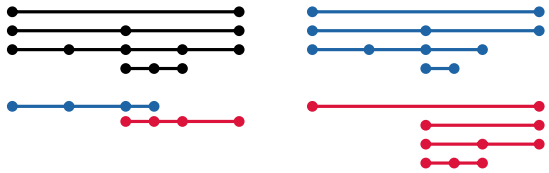
A `MultiDomainGrid` retains all capabilities of the underlying grid, including full support for h-adaptivity and MPI parallelism if these are provided by the host grid.

3.1.2 Grid Creation

As is typical for the meta grid approach, multi domain functionality is added to an existing grid by wrapping it in a `MultiDomainGrid`, which subsequently replaces the host grid in all further computations:

```
typedef MultiDomainGrid<
  HostGrid,
  FewSubDomainsTraits<
    HostGrid::dimension,
    maxSubDomains
    >
  > MDGrid;
bool support_level_views = true / false;
MDGrid md_grid(host_grid,support_level_views);
```

Here, the second template parameter is a policy class that can be used to customize internal data structures and algorithms for different application scenarios. Right now, it is possible two pick between three different policies, which are explained in detail in Section 3.2.2. The policy used here is optimized for situations where there is only a small number of subdomains, as is the case for the applications we are interested in. The maximum number of supported subdomains is controlled by the parameter `maxSubDomains`. Finally, it is possible to disable support for the hierarchic grid structure. If a simulation only uses the most-refined projection, the `LeafGridView` (as most simulations without h-adaptivity do), this optimization can substantially reduce the memory requirements of the grid.



◀**Figure 3.2** — Hierarchic subdomain construction from marked leaf grid: host grid hierarchy and marked leaf subdomains (left), resulting subdomain grid hierachies (right)

After creating the `MultiDomainGrid`, the host grid should not be accessed directly any more. In particular, bypassing the meta grid to perform any kind of operation that modifies the host grid will most likely result in undefined behavior.

3.1.3 Subdomain Setup

Subdomains are created implicitly by simply assigning grid cells to them. For this purpose, `MultiDomainGrid` provides an interface that closely resembles the existing DUNE interface for grid adaptation. Subdomains are always controlled through the `MultiDomainGrid`, which contains the new API. Note that modifying the subdomain layout and refining the grid are mutually exclusive processes; while one of the two is in progress, the other one cannot be started.

DUNE grids may actually contain a hierarchy of grid levels and allow user to access each of those levels individually in order to implement h-adaptive codes and solvers based on geometric multi grid. In `MultiDomainGrid`, a subdomain always comprises all levels of the grid; it is defined by marking cells on the *leaf grid* cells as belonging to the subdomain. Afterwards, this information is propagated up the grid hierarchy, but *not* back down again. For this reason, the refined children of a subdomain cell do not necessarily form a complete partition of the parent cell. The inter-level propagation of subdomain membership is demonstrated in Figure 3.2, which shows two subdomain grids constructed from two sets of marked cells on the leaf grid. This illustration also explains why it is sometimes not possible to make all children of a cell part of a subdomain: As the coarsest grid consists of only a single cell, doing so would cause both subdomains to grow and eventually contain the entire host grid, making it impossible to create any non-trivial subdomains on this host grid.

Subdomains are allowed to change their shape at any time and as often as required by the simulation, as long as there are no other grid modifications (load balancing, grid adaptation etc.) happening at the same time. Looking back at the two-phase / single-phase flow example shown in the introduction, this capability allows us to successively extend the two-phase flow region when the injected fluid starts to spread.

The programming interface for the subdomain marking process consists of a small number of `MultiDomainGrid` methods: Before we can begin to mark any cells, the process must be started by calling `md_grid.beginSubDomainMarking()`, which checks a number of prequisites and sets up the necessary data structures to

simultaneously manage the old and the new subdomain layout. Afterwards, we can iterate over the leaf grid and modify the subdomain structure:

```
for (const auto&& cell : cells(md_grid.leafGridView())) {
  md_grid.addToSubDomain(23,cell);       // add cell to subdomain 23
  md_grid.removeFromSubDomain(17,cell);  // remove cell from subdomain 17
  md_grid.removeFromAllSubDomains(cell); // remove cell from all subdomains
  md_grid.assignToSubDomain(42,cell);    // asssign cell to subdomain 42 only
}
```

The usage of these methods should be self-explanatory. Once the new layout has been completely built up, the actual switch over to that layout happens in three stages. This allows users to back up data stored in cells that are removed from a subdomain and to provide initial data when a cell is newly added to it. The procedure works similar to the way grid adaptation is handled by the DUNE grid interface:

- `md_grid.preUpdateSubDomains()` propagates the subdomain membership information to all codimensions and to the level grids. It also rebuilds the index maps based on the new subdomain layout. After this step, the user should do any required data projection between the old and the new subdomain state.
- A subsequent call to `md_grid.updateSubDomains()` activates the new subdomain layout, but retains all information about the old state, so that further user data transfer can occur.
- Once all data transfer has been completed, the bookkeeping information for the old layout is removed by calling `md_grid.postUpdateSubDomains()`.

3.1.4 Subdomain Usage

As already mentioned, subdomains are accessible as read-only DUNE grid objects. They support the complete DUNE interface with the exception of mutating operations like grid adaptation or load balancing. Note that `SubDomainGrid`s cannot be copied and must thus be obtained by reference from the `MultiDomainGrid`:

```
MDGrid::SubDomainGrid& sd_grid = md_grid.subDomain(42);
```

Afterwards, those `SubDomainGrid`s can be used like any other DUNE grid manager.

Inter-grid Data Transfer

Any operations that are restricted to a single subdomain can be performed using the corresponding `SubDomainGrid` in a completely transparent fashion, and our higher-level framework introduced in later chapters makes use of this feature to define function spaces on subdomains and support operations like solution output. In certain situations it is however necessary to access different levels of the meta grid hierarchy (e.g. to access the host grid). For this puprpose, both meta grid classes provide methods to convert between meta grid and host grid entities, shown

Listing 3.1 — Entity conversion methods in `MultiDomainGrid`

```
template<...>
class MultiDomainGrid {

  // MultiDomainGrid -> HostGrid
  const HostEntity&
  hostEntity(const MultiDomainEntity& e);

  HostEntityPointer
  hostEntityPointer(const MultiDomainEntity& e);

  // HostGrid -> MultiDomainGrid
  MultiDomainEntityPointer
  multiDomainPointer(const HostEntity& e);

  // SubDomainGrid -> MultiDomainGrid
  const MultiDomainEntity&
  multiDomainEntity(const SubDomainEntity& e);

  MultiDomainEntityPointer
  multiDomainEntityPointer(const SubDomainEntity& e);

};

template<...>
SubDomainGrid {

  // MultiDomainGrid -> SubDomainGrid
  SubDomainEntityPointer
  subDomainEntityPointer(const MultiDomainEntity& e);

};
```

in Listing 3.1. There is a corresponding set of methods for `Intersection`s, which we omit here for brevity.

The form assembly infrastructure of the multi domain framework does however take a different approach: It operates almost exclusively on the global `MultiDomainGrid` and directly queries the grid for the subdomains that a grid entity belongs to. This information is encoded in a `SubDomainSet`. For example, given a vertex object `vertex` on the leaf grid, its subdomain set is accessed by

```
const MDGridTraits::Codim<0>::SubDomainSet& subdomains =
  md_grid.leafGridView().indexSet().subDomains(vertex);
```

The `SubDomainSet` is a central data structure in our library (cf. Section 3.2). As the name implies, it stores a set of subdomains (identified by their index) and contains a comprehensive API to obtain information about its state and to make changes to the set. The following code snippet demonstrates how to use some important parts of that interface (we limit ourselves to the observer interface, as

subdomain sets should not be modified by users):

```
subdomains.size();                // number of contained subdomains
subdomains.contains(3);           // does the set contain subdomain 3?
subdomains.contains(subdomains2); // Is the set a superset of another set?
subdomains.empty();               // is the set empty?
for (auto subdomain: subdomains)  // iterator-based access to stored subdomains
  std::cout << subdomain << std::endl;
```

The actual implementation of the set depends on the policy chosen when the `MultiDomainGrid` was created; depending on the policy the exact implementation may also vary between codimensions for efficiency reasons. In keeping with DUNE's focus on stable, abstract interfaces, the exact implementation does not influence the public API shown above.

3.1.5 Subdomain Interface Extraction

Algorithms that calculate subdomain interactions must traverse the interface between those subdomains. We represent the elements of such an interface using the class `SubDomainInterface`, which is similar to a standard DUNE `Intersection`, but offers two additional methods

```
SubDomainIndexType subDomainInInside();
SubDomainIndexType subDomainInOutside();
```

for retrieving the subdomain indices of the two adjacent entities. Two iteration scenarios stand out as particularly useful and are directly supported by custom iterators in `MultiDomainGrid`:

Visiting the interface between two specific subdomains. Given two subdomains s_1 and s_2, this iterator will visit all intersections between entities e_i and e_j where e_i is contained in s_1 but not in s_2 and e_j is contained in s_2 but not in s_1. These iterators can be obtained using the grid methods

```
LeafSubDomainInterfaceIterator
leafSubDomainInterfaceBegin(SubDomainIndexType sd1,
                            SubDomainIndexType sd2);

LeafSubDomainInterfaceIterator
leafSubDomainInterfaceEnd(SubDomainIndexType sd1,
                          SubDomainIndexType sd2);
```

Note that the equivalent methods for level view iterators were omitted for brevity.

Visiting all subdomain interfaces in the grid. If an application needs to iterate over several subdomain interfaces, the iterators described above are not very efficient, because every pair of subdomains requires a full traversal of the underlying host grid. On the other hand, we can efficiently calculate the set of subdomain interfaces a given grid intersection belongs to: Given the intersection between two grid cells e_1 and e_2 belonging to the sets of

subdomains S_1 and S_2, respectively, let $D_1 = S_1 \setminus S_2$ and $D_2 = S_2 \setminus S_1$. Then the set of all subdomain intersections is given by the tensor product $D_1 \times D_2$. As outlined in Section 3.2.2, the per-entity set of subdomains is usually stored as a bitset in an integral type, making these set operations very efficient.

An efficient iteration scheme based on this algorithm that visits all subdomain interfaces of a `MultiDomainGrid` is implemented in the iterators obtained from the grid methods

```
1 LeafAllSubDomainInterfacesIterator
2 leafAllSubDomainInterfacesBegin();
3
4 LeafAllSubDomainInterfacesIterator
5 leafAllSubDomainInterfacesEnd();
```

and their level view equivalents. These iterators visit all interfaces between pairs of subdomains contained in the `MultiDomainGrid` and require only a single host grid traversal. If a single cell-cell intersection belongs to multiple subdomain interfaces, the iterator will return it once for each interface with different values for `subDomainInInside()` and `subDomainInOutside()`. It is thus often very beneficial to phrase application-level algorithms in such a way that all subdomain-subdomain related tasks can be handled in parallel using these iterators, yielding a performance benefit especially for larger numbers of subdomains that are small in comparison to the complete domain.

3.2 Implementation

One major problem related to meta grids in DUNE is the overhead associated with the wrapper layer around the underlying grid: While the DUNE grid interface makes heavy use of C++ template programming and is designed in a way that most of it can be optimized away by a good compiler, it was not designed with the possibility of creating meta grids in mind. Consequently, a certain amount of information duplication has to take place in the wrapper, making it expensive to "stack" several meta grids on top of each other.

Note that the next release of the DUNE core libraries will contain changes to the grid interface that largely mitigate this problem.

3.2.1 Design

The design and implementation of Dune-Multidomaingrid were inspired by the module Dune-Subgrid [58], but significantly expand on the concepts demonstrated there, as that module was built for a different purpose (geometric multi grid applications) and is only capable of tracking a single subdomain within a host grid. Dune-Subgrid was however the first DUNE meta grid and proved the feasibility of the concept: The grid interface can be used to proxy all unchanged functionality of the host grid by forwarding the user's method calls. That way,

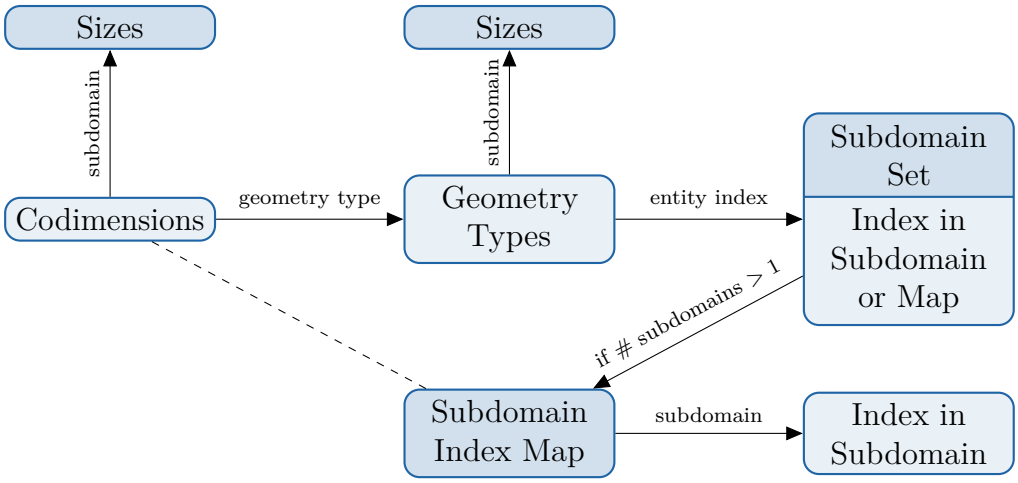


Figure 3.3 — Basic storage scheme for subdomain data in MultiDomainGrid

meta grid implementors can focus their efforts on the additional features they want to add to the host grid and combine these with the vastly different feature sets of the different DUNE grid implementations.

Since individual subdomains simply represent a subset of the entities contained in the host grid, the only functionality that needs to be changed for the subdomain tracking are those parts of the grid interface concerned with the entity complex of the host grid. This includes entity and intersection iterators, entity counts and index maps, but not ID maps, as the IDs generated by the host grid are still unique for each subdomain and can be reused. Implementing any of these features requires a subdomain set map $\sigma : E \to \mathcal{S}$ that associates every entity $e \in E$ with the set of subdomains $s \subseteq S$ it is a member of. Here, S is the set of all possible subdomains and $\mathcal{S} := \mathcal{P}(S)$ denotes the set of all possible subdomain combinations. At the implementation level, σ is realized by storing a `SubDomainSet` instance for every entity of the `MultiDomainGrid`. The grid interface makes it possible to attach data to the entities in a `GridView` (essentially, either a level grid or the leaf projection) by means of the attached index sets (cf. Section 2.3.2), which provide a consecutive index range for each type of grid entity. Using these index maps, we can story the per-entity subdomain information in flat arrays, allowing for efficient data access.

As a result, the subdomain membership information is closely tied to the `IndexSet` component. We thus store this information as part of the `MultiDomainGrid` index set, which wraps the index set of the underlying host grid. In addition, the `IndexSet` is also one of the few locations in the DUNE grid interface that can easily be extended with additional user-visible interface methods; most grid components must be wrapped in a facade layer that only exposes the officially mandated APIs as per the grid interface specification, a limitation that would be difficult to circumvent.

While it is possible to devise a number of different storage schemes for the subdomain, the one implemented by `MultiDomainGrid` is depicted in Figure 3.3. The two containers for codimension and geometry type are due to the fact that the index maps of the grid interface are defined in terms of the geometry type (essentially, the reference element of a grid entity), i.e. an index map I_{gt} for entities of geometry type gt is a map $E_{gt} \to [0, |E_{gt}| - 1]$. It is thus necessary to create a distinct data structure for each geometry type contained in the grid. This data structure is an array of tuples (s, i), where s denotes the set of subdomains the entity belongs to, and i represents an index. The precise meaning of i depends on the cardinality of s: if $|s| = 1$, it represents the index of the entity within the single subdomain it belongs to. Otherwise, it refers to an entry in a second array which contains a map $\lambda_e : s \to I$.

3.2.2 Storage Backends

The algorithm behind the storage strategy described above is currently fixed, but it can nevertheless be influenced by choosing different implementations for several map and set containers. In particular, it is possible to specify the types of the containers shaded in darker blue in Figure 3.3. Right now, there are three pre-defined policies that can be picked when creating a `MultiDomainGrid`:

- `FewSubDomainsTraits`: This policy is optimized for regular multi-physics problems. It allows for up to 64 subdomains that may overlap in an arbitrary fashion. With this policy, entity subdomain membership tests and of subdomain entity index are very fast and of $\mathcal{O}(1)$ complexity. The hard limit of 64 is due to the fact that the subdomain set is stored as a bitmask in an integral type.
- `ArrayBasedTraits`: An alternative policy designed for large numbers of subdomains. It does not place any intrinsic limit on the maximum number of subdomains, but limits the number of subdomains a single entity can belong to. Both subdomain membership testing and subdomain index lookup require a single binary search of a sorted array that contains the local set of subdomains and are thus of complexity $\mathcal{O}(\log |s_e|)$. Note that while this policy allows for arbitrarily many subdomains, the user still has to pick a per-grid maximum, which is passed as a template parameter to the traits class. The grid needs to know about this number because it has to allocate storage for subdomain-specific data like entity counts.
- `DynamicSubDomainCountTraits`: This policy is a slightly more flexible version of the `ArrayBasedTraits` that only requires the user to specify the maximum number of subdomains at run time. This additional flexibility does however come at the price of a small performance penalty: The grid now has

to allocate dynamic memory for storing per-subdomain information, introducing an additional pointer indirection relative to the `ArrayBasedTraits`, which store this information in statically allocated arrays.

3.2.3 Efficiency

The entity and intersection iterators for subdomains are implemented on top of the host iterators. They always iterate over the complete host grid and simply skip entities that are not contained in the subdomain. Moreover, they modify the intersection type for intersections that are in the interior of the `MultiDomainGrid`, but on the outer boundary of a subdomain. This implementation strategy causes iteration time over a subdomain to scale linearly with the size of the complete domain, albeit with a small constant. While this may create some overhead when iterating over very small subdomains, the problem can mostly be avoided by coalescing subdomain iterations into a single pass over the underlying `MultiDomainGrid`. This approach is taken in Dune-Multidomain, our PDELab extension module based on `MultiDomainGrid`, where only grid I/O operations employ the inefficient iteration pattern.

Grid adaptation is handled transparently by the implementation. It is possible to place refinement marks on both the `MultiDomainGrid` as well as any `SubDomainGrid`, but the actual grid transformation can only be initiated by the `MultiDomainGrid`, a choice that was made to explicitly emphasize the fact that there is only a single host grid and as a result, refining one subdomain will also affect other, overlapping subdomains.

In order to evaluate the runtime and memory overhead of `MultiDomainGrid`, we took a simple example program from PDELab that solves the Poisson equation in 2D on the unit square using a mix of Dirichlet and Neumann boundaries. This program was modified to run either directly on the host grid, on the `MultiDomainGrid` or on a `SubDomainGrid` spanning the complete domain. This way, all three program variants solve the exact same problem in an identical fashion, which allows for a good assessment of the overhead imposed by wrapping the host grid and by using a grid defined on a subdomain, respectively. We ran the benchmark using both a structured (`YaspGrid`) and an unstructured (`ALUSimplexGrid`) host grid to investigate whether our module exhibits a different behavior on those two types of grids. The grids for the benchmark were generated by starting with a single square (or two triangles in the case of `ALUSimplexGrid`) covering the unit square and iteratively refining those macro grids. All results were obtained by running the simulations 10 times on hardware configuration B.1 and averaging the numbers obtained from the individual runs.

We assessed the runtime overhead of our grid by timing several standard PDELab operations which all involve a grid iteration, but vary in the computational effort per grid cell, ranging from the very fast grid function space setup to the evaluation of the Jacobian by numerical differentation. The results of the comparison can be

Host Grid	YaspGrid				ALUSimplexGrid			
Grid Size	262144				524288			
	t_{host}[s]	$\frac{t_{MD}}{t_{host}}$	$\frac{t_{SD}}{t_{host}}$	$\frac{t_{SD}}{t_{MD}}$	t_{host}[s]	$\frac{t_{MD}}{t_{host}}$	$\frac{t_{SD}}{t_{host}}$	$\frac{t_{SD}}{t_{MD}}$
GFS Setup	0.177	1.19	2.36	1.99	0.720	1.55	3.18	2.05
Constraints	0.313	1.46	4.66	3.19	0.821	1.62	4.09	2.53
Pattern	1.14	1.04	1.21	1.16	1.38	1.16	1.64	1.42
Residual	1.00	1.02	1.55	1.52	2.04	1.15	1.79	1.56
Jacobian	3.56	1.03	1.09	1.05	5.11	1.05	1.17	1.11

Table 3.1 — Performance comparison between MultiDomainGrid (t_{MD}), associated SubDomainGrid (t_{SD}) and underlying host grid (t_{host}) for common PDELab operations. Grids were refined 9 times.

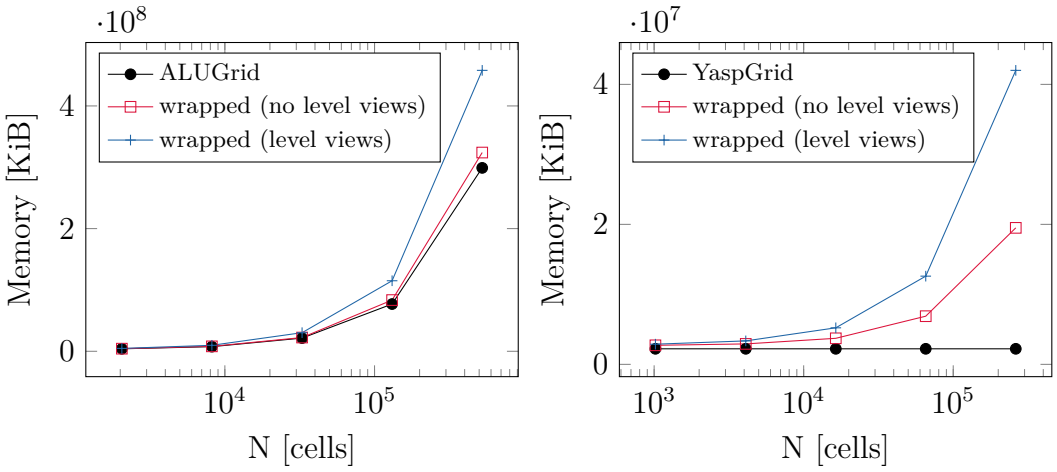


Figure 3.4 — Memory usage of MultiDomainGrid and underlying host grid for ALUGrid (unstructured) and YaspGrid (structured).

found in Figure 3.1. Those results clearly show a noticeable performance overhead which might be reduced by further optimization of the wrapper implementation. In particular, the `SubDomainGrid`-based variant exhibits a disproportionate runtime increase, which is linked both to the fact that it is implemented as a second meta grid stacked on top of the `MultiDomainGrid`, but also to the additional subdomain membership checks required during the iteration. In general, the perfomance penalty is more pronounced for simple and fast operations, making the grid in its current state more suited to numerical schemes that involve a moderate to large computational effort per cell.

The additional memory requirements of `MultiDomainGrid` are illustrated in Figure 3.4. The memory usage of the programs was measured directly after grid creation (for the host grid) and after creating the subdomains (for `MultiDomainGrid`). Our

grid necessarily changes the memory usage characteristics when used on top of a structured grid (which uses the same amount of memory independent of the grid size). This is to be expected – the subdomains we provide are not necessarily structured anymore, so we lose the optimization opportunities exploited by a structured host grid. On the other hand, the scaling behavior of an unstructured grid with respect to its memory requirement remains unchanged (essentially linear in the number of grid entities). Unfortunately, the total amount of extra memory required is still rather substantial. In order to mitigate this problem, it is possible to reduce the storage requirements of a `MultiDomainGrid` in two ways:

Remove support for unused codimensions. For example, a Finite Volume or Discontinuous Galerkin discretization only requires entities of codimension 0. In this case, it is possible to selectively deactivate unused entities (edges, faces) in the `SubDomainGrid`s. Trying to access such an entity in a `SubDomainGrid` will then result in a compile time error. The corresponding entity on the `MultiDomainGrid` can still be used, but the grid will no longer track its subdomain memberships. Note that a `SubDomainGrid` must always contain both cells and vertices. This optimization can be enabled by setting a template parameter on the policy class of the `MultiDomainGrid`.

Deactivate level grid views. Non-adaptive codes will normally only access the leaf grid view of a grid. If a program never accesses the level grid views of the subdomains, support for them can be removed at run time when creating the `MultiDomainGrid`. Depending on the number of levels contained in the host grid, this optimization may yield a massive reduction in memory usage, but even for a completely unrefined grid, we are able to approximately cut the memory requirements in half. As can be seen in Figure 3.4, deactivating the level grid views reduces the memory overhead for unstructured grids to a mostly negligible amount.

These optimizations mainly affect the memory requirements of the module and do not really affect the run time performance during normal grid operations. They do, however, reduce the time it takes to rebuild the subdomain information after changing the subdomain layout, as there are fewer entities that need to be tracked and assigned per-subdomain indices. Substantial improvements to the runtime efficiency of the module require modifications to the DUNE grid interface. Unfortunately, some of those changes cannot be made in a backwards compatible way, but recently the DUNE developers agreed to introduce them as part of a major new release that will be allowed to break backwards compatibility.

In the context of our multi domain framework, we are able to sidestep most of the performance problems: They are mostly associated with the `SubDomainGrid`s, and as we will see in Chapter 7, all performance-critical operations are implemented directly on top of the `MultiDomainGrid`; the `SubDomainGrid`s are only used for a small number of tasks like I/O, where performance is not an issue.

CHAPTER 4

Mathematical Framework for General Multi Physics Problems

In this chapter we define a mathematically rigorous framework for the definition of multi domain problems and introduce our software library Dune-Multidomain which implements this mathematical framework by extending the PDE solver toolbox PDELab. After introducing the problem setting and its associated challenges via several example problems, we start by reiterating the basic PDELab principle of recursively composing complex function spaces from component spaces before introducing multi domain specific extensions like support for function spaces and residuals that are only defined on part of the overall simulation domain. Throughout the chapter we will refer back to the initial example problems and show how they map to the components of our framework, both at a mathematical level and with code examples that demonstrate the general usage of our implementation.

The purpose of this framework is to provide a precise notation for describing *discrete* residuals (and Jacobians) which we want to assemble on a given set of related meshes with associated *discrete* function spaces. Our framework thus limits itself to the description of discrete problems. While it is entirely possible to extend its abstractions to describe the problems at a continuous level, the required effort is beyond the scope of this work and is better placed in a more theoretical treatise focused on mathematical analysis.

Our framework relies on Dune-Multidomaingrid for the spatial discretization of the domain and its division into subdomains, so there will be occasional references to `MultiDomainGrid`-specific terminology like subdomain sets introduced in the previous chapter.

Remark 4.1. As in the previous chapters, we partially omit the namespace scope of

the classes and functions of our framework in the code examples; specifically, we omit both `Dune::` and `Dune::PDELab::` prefixes. Note that all of the new functionality introduced in this chapter lives in the namespace `Dune::PDELab::MultiDomain` and can thus be recognized by the prefix `MultiDomain::`.

4.1 Introduction

Dune-Multidomain has been developed as an add-on module for PDELab and is compatible with the current 2.0 release branch of that software. Moreover, it relies on Dune-Multidomaingrid for subdomain information and is compatible with its 2.3 release series. Dune-Multidomain is free software and available under the same licence as the DUNE core modules (the GNU General Public Licence with a special runtime exception, for further details see [40]).

The following description is based on version 2.0.1 of the library, which can be downloaded from [91] or directly from the source code repository at [92]. It requires the 2.3.1 release of the DUNE core modules.

4.2 Problems with Multiple Variables

A multi physics problem usually involves more than a single quantity of interest – normally, we are interested in multiple variables (e.g. pressure, concentration, velocity,...) and thus the solution space U will be vector-valued. For our purposes, we assume that U can always be written as a tensor product of elementary spaces, so for a function $u = (u_1, u_2, \dots, u_n) \in U$, the space can be written as

$$U = U_1 \times U_2 \times \cdots \times U_n. \tag{4.1}$$

with $U_i = \mathcal{S}(\Omega_i)$ some Sobolev space on an associated open domain $\Omega_i \subset \mathbb{R}^d$. Note that this definition allows each variable to have a distinct spatial domain. Moreover, each of those elementary spaces U_i can still be vector- or tensor-valued, e.g. for $H(\mathrm{curl})$ or $H(\mathrm{div})$ spaces.

In the following, we introduce a number of prototypical multi domain problems and discuss what kind of features our assembly framework needs to support these problems. Afterwards, we give a high-level overview of the framework; in many places, its interfaces and design choices will be motivated by examples and code snippets that refer back to the example problems.

4.2.1 Two Domain Poisson Problem

We begin with a simple two domain Poisson problem on nonoverlapping subdomains with Dirichlet-Neumann coupling conditions on the interface Γ_C and homogeneous Dirichlet boundary conditions on the outer boundary as depicted in Figure 4.1:

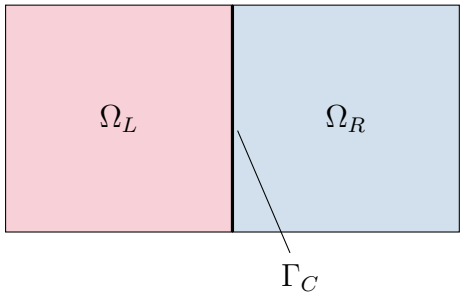


Figure 4.1 — Two coupled Poisson problems on adjacent domains

$$\begin{aligned} -\Delta u_i &= f \quad \text{in } \Omega_i, \quad i \in \{L, R\}, \\ u_i &= 0 \quad \text{on } \partial\Omega_i \setminus \Gamma_C, \\ u_L &= u_R \quad \text{at } \Gamma_C, \\ (\nabla u_L) \cdot \boldsymbol{n} &= (\nabla U_R) \cdot \boldsymbol{n} \quad \text{at } \Gamma_C, \end{aligned} \tag{4.2}$$

where f denotes a source / sink term. Note that this problem is equivalent to a standard Poisson problem on the combined domain $\Omega = \Omega_L \cup \Omega_R \cup \Gamma_C$:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{4.3}$$

This very reduced setting already allows us to extract a number of features our assembly framework must support:

- It should be possible to use different discretization schemes for the two domains and e.g. couple a continuous Galerkin discretization with a DG scheme.
- We want to be able to support meshes that are nonconforming on the coupling interface Γ_C.
- The problem clearly separates into two subproblems, along with coupling terms. Accordingly, we want to be able to either apply a global, monolithic solver to the overall problem or employ a scheme that iterates between the two subproblems, e.g. a Dirichlet-Neumann iteration.

If we assume a continuous Galerkin approach for each subdomain and meshes $\mathcal{T}_L$ and $\mathcal{T}_R$ for Ω_L and Ω_R, respectively, the discrete residual r for the overall problem

becomes

$$
\begin{aligned}
r\Big((u_L,u_R),(v_L,v_R)\Big) = \quad & \sum_{T\in\mathcal{T}_L}\int_T \nabla u_L\cdot\nabla v_L\,\mathrm{d}x + \sum_{T\in\mathcal{T}_L}\int_T f v_L\,\mathrm{d}x && \text{(4.4a)}\\
+ & \sum_{T\in\mathcal{T}_R}\int_T \nabla u_R\cdot\nabla v_R\,\mathrm{d}x + \sum_{T\in\mathcal{T}_R}\int_T f v_R\,\mathrm{d}x && \text{(4.4b)}\\
+ & \sum_{\tau^{(c)}\in\mathcal{T}_{\Gamma_C}}\int_{\tau^{(c)}} [u][v]\,\mathrm{d}s && \text{(4.4c)}\\
+ & \sum_{\tau^{(c)}\in\mathcal{T}_{\Gamma_C}}\int_{\tau^{(c)}} [\nabla u]\cdot[\nabla v]\,\mathrm{d}s, && \text{(4.4d)}
\end{aligned}
$$

where $[u] = u_L - u_R$ denotes the jump of a function across a cell intersection and $\{u\} = \frac{1}{2}(u_L + u_R)$ the average. Here, (4.4a) and (4.4b) correspond to the individual Poisson problems on each subdomain (with associated mesh $\mathcal{T}_i$), while (4.4c) and (4.4d) provide the coupling between those two problems. Note the additional mesh $\mathcal{T}_{\Gamma_C}$: In general, generating this mesh and calculating its topological relationships to the subdomain meshes proves to be one of the major implementation challenges. In our setting, we rely on `SubDomainGrid` to supply this crucial information, which allows us to exploit the known global topology of the host grid.

4.2.2 Stokes-Darcy Flow

While the simple Poisson-Poisson problem introduced above works well as a vehicle for understanding the basic mathematical and implementation issues of a multi domain simulation, real-world applications will typically be more complicated in that they feature different variables and equations in each subdomain. As an example of such a problem, we consider the problem of stationary coupled flow in a free-flow domain and a porous medium. Figure 4.2 shows an example of such a problem along with a solution that was computed using our framework.

The free flow in this problem is described by the standard (Navier-)Stokes equations, which require separate variables $\boldsymbol{v}$ for velocity and p for fluid pressure. In the following, we neglect the convective flow component and restrict ourselves to Stokes flow, which keeps the overall problem linear and is an acceptable simplification if we only consider creeping flow. For this example, we follow the formulation in [29]:

$$
\begin{aligned}
\nabla\cdot\Big(2\mu\mathbb{D}(\boldsymbol{v}) - p\mathbb{I}\Big) &= \boldsymbol{f}_S \text{ in } \Omega_S,\\
\nabla\cdot\boldsymbol{v} &= 0 \text{ in } \Omega_S,\\
\boldsymbol{v} &= \boldsymbol{g_{S,D}} \text{ on } \partial\Omega_S\setminus\Gamma_C.
\end{aligned}
$$

Here, μ is the viscosity of the fluid, $\boldsymbol{f}_S$ denotes a possible external force like gravity, $\mathbb{D} = \frac{1}{2}\Big(\nabla\boldsymbol{v} + (\nabla\boldsymbol{v})^T\Big)$ the symmetric deformation tensor and $\mathbb{T} = 2\mu\mathbb{D}(\boldsymbol{v}) - p\mathbb{I}$ the

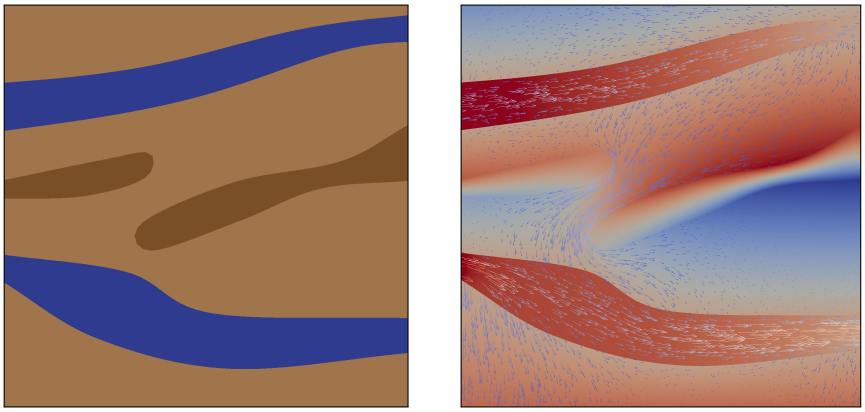


Figure 4.2 — Simulation domain and solution for a Stokes-Darcy problem. Left: Domain with two free-flow channels from left two right and two areas with very low permeability in the porous medium. Right: Pressure (color) and velocity (glyph) fields of an example simulation.

stress tensor. For simplicity, we assume all Dirichlet boundary conditions on the outer domain boundaries.

For the porous medium, we do not resolve the pore scale, but use an upscaled, homogenized model of Darcy flow with the hydraulic head $\phi = \frac{p}{\varrho g} + z$ as the single primary variable. In this model, flow always occurs in direction of the negative gradient of ϕ and is proportional to its absolute value:

$$\begin{aligned} -\nabla \cdot \Big(\boldsymbol{K} \nabla \phi\Big) &= f_D \text{ in } \Omega_D, \\ \phi &= g_{D,D} \text{ on } \partial\Omega_D \setminus \Gamma_C, \end{aligned}$$

where $\boldsymbol{K}$ denotes the permeability tensor and f_D a possible source term. The hydraulic head ϕ is mostly identical to the fluid pressure normalized by density, but takes into account the effect of gravity.

The two domains are usually coupled by a set of conditions derived experimentally by Beavers and Joseph [21]. They were later simplified by Saffman [111] and Jones [75] and are typically called the Beavers-Joseph(-Saffman) conditions, depending on whether the full or the simplified conditions are used. More recently, Jäger and Mikelić [74] presented a theoretically motivated derivation of the equations based on an analytical homogenization framework. In our notation, the equations read

$$\boldsymbol{v_S} \cdot \boldsymbol{n}_{SD} = -\Big(\boldsymbol{K} \nabla \phi\Big) \cdot \boldsymbol{n}_{SD}, \tag{4.5a}$$

$$-\boldsymbol{n}_{SD}^T \mathbb{T}(\boldsymbol{v}) \boldsymbol{n}_{SD} = g(\phi - z), \tag{4.5b}$$

$$-P_\tau(\mathbb{T}(\boldsymbol{v})\boldsymbol{n}_{SD}) = \frac{\alpha\mu\sqrt{3}}{\sqrt{\operatorname{tr}(\mu \boldsymbol{K}/g)}} P_\tau(\boldsymbol{v} + \boldsymbol{K}\nabla\phi). \tag{4.5c}$$

Here, $\boldsymbol{n}_{SD}$ denotes the interface normal pointing from the Stokes to the Darcy domain and $P_\tau(\boldsymbol{x}) = \boldsymbol{x} - (\boldsymbol{x}, \boldsymbol{n}_{SD})\boldsymbol{n}_{SD}$ the projection operator into the tangential

plane orthogonal to $\boldsymbol{n}$. With this in mind, we observe that the sum of (4.5b) and (4.5c) yields $\mathbb{T}(\boldsymbol{v})\boldsymbol{n}_{SD}$, which is the natural boundary condition of the Stokes problem.

In the following, we present a discrete formulation of this problem based on a continuous Galerkin discretization of the Stokes part and a DG scheme for the porous medium. Here, the DG approach has the advantage of being locally mass-conservative; moreover, DG schemes are better suited to heterogeneous parameter fields like the large jump in permeability shown in Figure 4.2, where the permeability differs by a factor of 10^5 between the two differently shaded areas inside the porous medium. Due to the large number of terms in the residual, we have split it into three separate parts for the two subdomains and the coupling conditions. The Stokes residual r_S in this model is given by

$$\begin{aligned} r_S\Big((\boldsymbol{v},p),(\boldsymbol{w},q)\Big) = & \sum_{T\in\mathcal{T}_S}\int_T\Big(p\mathbb{I}-2\mu\mathbb{D}(\boldsymbol{v})\Big)\cdot\nabla\boldsymbol{w}\,\mathrm{d}x \\ & + \sum_{T\in\mathcal{T}_S}\int_T \boldsymbol{f}_S\cdot\boldsymbol{w}\,\mathrm{d}x + \sum_{T\in\mathcal{T}_S}\int_T(\nabla\cdot\boldsymbol{v})q\,\mathrm{d}x. \end{aligned} \tag{4.6}$$

For the porous medium, we employ a standard Symmetric Interior Penalty Galerkin (SIPG) discretization. As we are not interested in the underlying analysis, we skip the lengthy derivation of this scheme and just state the resulting residual form r_D; for further information see e.g. [49]:

$$r_D(\phi,\psi) = \sum_{T\in\mathcal{T}_D}\int_T(\boldsymbol{K}\nabla\phi)\cdot\nabla\psi\,\mathrm{d}x \tag{4.7a}$$

$$+ \sum_{\tau\in E(\mathcal{T}_D)}\int_\tau\{(\boldsymbol{K}\cdot\boldsymbol{n}_\tau)\cdot\nabla\phi\}\cdot[\psi]\,\mathrm{d}s \tag{4.7b}$$

$$+ \sum_{\tau\in E(\mathcal{T}_D)}\int_\tau[\phi]\cdot\{(\boldsymbol{K}\cdot\boldsymbol{n}_\tau)\cdot\nabla\psi\}\,\mathrm{d}s \tag{4.7c}$$

$$+ \sum_{\tau\in E(\mathcal{T}_D)}\int_\tau\frac{\alpha}{h_\tau}[\phi][\psi]\,\mathrm{d}s \tag{4.7d}$$

$$- \sum_{\tau\in B(\mathcal{T}_D)}\int_\tau((\boldsymbol{K}\cdot\boldsymbol{n}_\tau)\cdot\nabla\phi)\psi\,\mathrm{d}s \tag{4.7e}$$

$$- \sum_{\tau\in B(\mathcal{T}_D)}\int_\tau(\phi-g)((\boldsymbol{K}\cdot\boldsymbol{n}_\tau)\cdot\nabla\psi)\,\mathrm{d}s \tag{4.7f}$$

$$+ \sum_{\tau\in B(\mathcal{T}_D)}\int_\tau\frac{\alpha}{h_\tau}(\phi-g_{D,D})\psi\,\mathrm{d}s, \tag{4.7g}$$

where $E(\mathcal{T}_D)$ denotes the set of all cell intersections in the interior of the Darcy subdomain and $B(\mathcal{T}_D)$ the set of all cell intersections on the outer boundary, but not on the coupling interface. α is a problem- and discretization-dependent scaling parameter for the penalty term and h_τ is the intersection diameter. This example

also demonstrates that PDELab is not restricted to "standard" continuous FE methods, but can just as easily be used for nonconforming methods like Finite Volumes (FVs) or in this case DG.

Finally, for the coupling residual r_C we multiply the first condition (4.5a) with a test function from the Stokes domain and the remaining two conditions (4.5b) and (4.5c) with the normal and tangential part of a velocity test function from the Stokes domain, respectively:

$$\begin{aligned} r_C &= r_C\Big(((\boldsymbol{v}, p), \phi), ((\boldsymbol{w}, q), \psi)\Big) \\ &= \sum_{\tau \in \mathcal{T}_{\Gamma_C}} \int_\tau (\boldsymbol{v} \cdot \boldsymbol{n}_{SD}) \psi \, \mathrm{d}s && \text{(4.8a)} \\ &\quad + \sum_{\tau \in \mathcal{T}_{\Gamma_C}} \int_\tau g(\phi - z)(\boldsymbol{w} \cdot \boldsymbol{n}_{SD}) \, \mathrm{d}s && \text{(4.8b)} \\ &\quad + \sum_{\tau \in \mathcal{T}_{\Gamma_C}} \int_\tau \frac{\alpha\mu\sqrt{3}}{\sqrt{\mathrm{tr}(\mu \boldsymbol{K}/g)}} P_\tau(\boldsymbol{v} + \boldsymbol{K}\nabla\phi) \cdot P_\tau(\boldsymbol{w}) \, \mathrm{d}s && \text{(4.8c)} \end{aligned}$$

The coupling residual r_C essentially prescribes a natural (Neumann) boundary condition for each subdomain on the coupling interface Γ_C.

Looking at this example, we can identify a number of additional requirements for our framework:

- For realistic problems, the coupling residual r_C will not be symmetric and we need to be able to specify the orientation of the coupling interface (as evidenced above by the order of the arguments to r_C and the direction of the normal $\boldsymbol{n}_{SD}$ pointing from the Stokes to the Darcy domain).
- For realistic problems and discretizations, the assembly of each subproblem (r_S and r_D) is often already a rather complicated process; it should be possible to reuse implementation building blocks from existing simulations for individual subdomains.
- Individual subproblems may already exhibit a complex internal structure, as evidenced by the Stokes subproblem in this example. In general, we cannot expect all components that we want to combine into a multi domain simulation to lay out this structure in a uniform way (consider e.g. the Stokes function space, which can either be written as $\boldsymbol{V} \times P$ or $P \times \boldsymbol{V}$). Our framework thus needs a mechanism to convert data on the fly between these different representations.

4.2.3 Two Model Two-Phase Flow Problem

In addition to the "classical" multi domain topologies with disjoint subdomains, our framework is also capable of handling problems with overlapping domains. As an example, consider the problem of CO_2 injection into underground reservoirs as

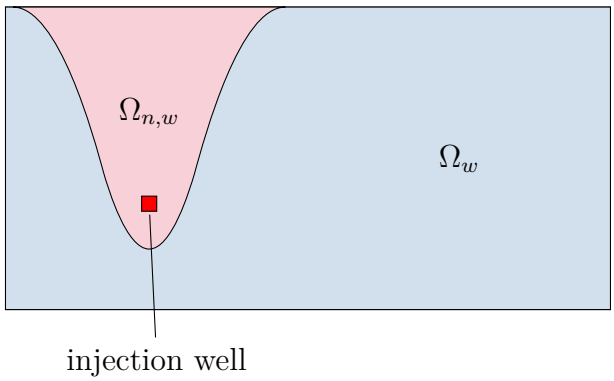


Figure 4.3 — Layout of CO_2 injection simulation with separate models for regions with / without CO_2. The water pressure p is defined in Ω_w, which spans the whole simulation domain, while the CO_2 saturation is only modeled in the area $\Omega_{n,w}$ surrounding the injection well.

mentioned in the introduction. In the following, we will consider a greatly simplified version of this problem as depicted in Figure 1.2. In order to avoid obfuscating the multi domain specific challenges with the complexities of a full CO_2 / water model, we restrict ourselves in the following to a model of basic two-phase flow of a wetting phase w (e.g. water) and a non-wetting phase n (e.g. oil) in a porous medium, which is described by the Darcy equations for two-phase flow:

$$\partial_t(\phi\rho_\alpha S_\alpha) - \nabla\Big(\boldsymbol{K}\frac{k_{r,\alpha}}{\mu_\alpha}\rho_\alpha(\nabla p_\alpha - \rho_\alpha g)\Big) = \rho_\alpha q_\alpha \quad \text{in } \Omega\times\Sigma, \quad \alpha\in\{w,n\}, \tag{4.9a}$$

$$p_\alpha = g_\alpha \quad \text{on } \partial\Omega, \quad \alpha\in\{w,n\}, \tag{4.9b}$$

$$p_\alpha(\cdot,t_0) = p_{\alpha,t_0}, \tag{4.9c}$$

$$p_n - p_w = p_c(S_w), \tag{4.9d}$$

$$S_w + S_n = 1, \tag{4.9e}$$

where ϕ denotes the porosity, p_c the capillary pressure, $\boldsymbol{K}$ the absolute permeability tensor, $S_\alpha \in [0,1]$ the saturation of phase α and its saturation-dependent relative permeability. ρ_α is the density of phase α, μ_α its viscosity, q_α a source / sink term and g the gravitational acceleration. This is a system of 4 equations for the 4 unknowns p_w, p_n, s_w and s_n. In order to solve this problem, we have to pick a model for the capillary pressure and the relative permeability. In the following, we choose the model by Brooks and Corey [24], who experimentally derived the equations

$$p_c = p_e S_w^{-1/\lambda}, \tag{4.10a}$$

$$k_{r,w} = S_w^{\frac{2+3\lambda}{\lambda}}, \tag{4.10b}$$

$$k_{r,n} = S_n^2(1-(1-S_n)^{\frac{2+\lambda}{\lambda}}), \tag{4.10c}$$

where p_e denotes the entry pressure, λ is the pore size distribution index and γ_α are empirical, problem specific parameters. Note that this model requires a renormalization of p_c to allow for a vanishing wetting phase. We then use the *constitutive relationships* (4.9d) and (4.9e) together with the capillary pressure model (4.10a) to eliminate the wetting phase saturation S_w and the nonwetting phase pressure p_n from (4.9), which yields

$$\partial_t\Big(\phi\rho_w(1-S_n)\Big) - \nabla\Big(\boldsymbol{K}\frac{k_{r,w}}{\mu_w}\rho_w(\nabla p_w - \rho_w g)\Big) = \rho_w q_w, \tag{4.11a}$$

$$\partial_t(\phi\rho_n S_n) - \nabla\Big(\boldsymbol{K}\frac{k_{r,n}}{\mu_n}\rho_n\Big(\nabla(p_w + p_e(1-S_n)^{-1/\lambda}) - \rho_n g\Big)\Big) = \rho_n q_n, \tag{4.11b}$$

$$p_w = g_w \text{ on } \partial\Omega, \tag{4.11c}$$

$$S_n = g_n \text{ on } \partial\Omega, \tag{4.11d}$$

$$p_w(\cdot, t_0) = p_{w,t_0}, \tag{4.11e}$$

$$S_n(\cdot, t_0) = S_{n,t_0}, \tag{4.11f}$$

This is the standard pressure-saturation formulation of a two-phase flow problem in a porous medium. Due to the nonlinearity of p_c and $k_{r,\alpha}$, this system of PDEs is highly nonlinear and requires significant effort to solve.

In contrast, the flow of a single phase in a porous medium can be described with the Darcy model given by

$$\partial_t(\phi\rho_w p_w) - \nabla\Big(\boldsymbol{K}\frac{\rho_w}{\mu_w}(\nabla p_w - \rho_w g)\Big) = \rho_w q_w, \tag{4.12}$$

which is a much simpler, linear diffusion-type equation.

Looking back at the scenario outlined in Figure 4.3, it seems advantageous to only solve the full two-phase model (4.11) in the small (compared to the overall simulation domain) area surrounding the well that actually contains the second phase and switch to the much simpler single-phase model (4.12) for the remainder of the domain.

Typically, both of these models will be implemented with a locally mass conserving discretization like FV or DG. The resulting residuals are similar to the Darcy residual from the previous example; we thus refrain from explicitly stating them here.

This scenario reveals several new requirements for our framework:

- In the previous examples, the integration domains of the subproblem-specific residuals coincided with the domain of the underlying variables. This no longer holds here: The water pressure variable p_w exists on the whole integration domain Ω, but the one-phase residual corresponding to (4.12) is only defined on the smaller region $\Omega \setminus \Omega_{n,w}$. We thus need to be able to restrict the integration domain of a subproblem residual to a subset of the domain of its associated variables.

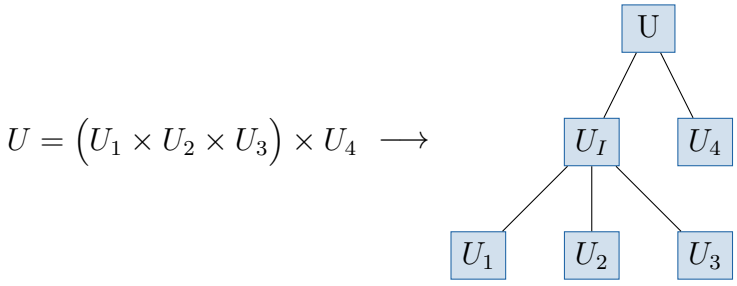


Figure 4.4 — Composite function space as recursive tree

- As the time in the simulation advances, $\Omega_{n,w}$ will change in size, so our framework needs to cope with subdomains that are modified during the simulation and provide support for migrating data (solutions, boundary conditions etc.) between different subdomain layouts.

4.3 Hierarchical Construction of Composite Function Spaces

As mentioned in Section 2.1.4, the function spaces for problems with multiple variables are the algebraic product of the elementary spaces for each variable:

$$U = U_0 \times \cdots \times U_{n-1},$$

a process that we can repeat multiple times to recursively create more complicated, structured spaces. Mathematically, this structure corresponds to the grouping of terms in the algebraic product using parentheses, e.g.

$$U = U_1 \times U_2 \times U_3 \times U_4 = \left(U_1 \times U_2 \times U_3\right) \times U_4$$

and becomes clearer when looking at the corresponding *expression tree*, which allows us to interpret U as a function space tree as shown in Figure 4.4. This figure also demonstrates an important effect that occurs when mapping the nested tensor product to a tree structure: Each group of function spaces is represented by an inner node of the tree (U_I in this example). These artificially introduced objects will be very useful in our mapping of mathematical concepts to software, as they provide a handle for arbitrary subtrees of the overall function space. These parts can be passed to building blocks like operator implementations for individual subproblems without making those building blocks aware of the overall problem structure, a critical requirement to enable the reuse of these (single physics, single domain) components in a larger multi domain context.

The mathematical tree notation directly maps to the default implementation in PDELab. For example, a composite space with the children `GFS0`, `GFS1` and `GFS2` is created as

```
typedef CompositeGridFunctionSpace<
  OrderingTag,
  VectorBackend,
  GFS0,GFS1,GFS2
  > CompositeSpace;
CompositeSpace composite_space(gfs0,gfs1,gfs2);
```

Here, the first two template arguments are used to control the mapping of DOFs to vector entries, see Chapter 6 for further information.

From a theoretical point of view, this implementation is sufficient for all types of composite spaces, but the interface becomes increasingly unwieldy if the number of child spaces becomes too large (e.g. when modeling a reaction problem with 20 or 30 reactants). For this reason, PDELab contains a second type of composite space, the `PowerGridFunctionSpace`:

```
typedef PowerGridFunctionSpace<
  ChildGFS,N,
  OrderingTag,
  VectorBackend
  > PowerSpace;
PowerSpace power_space(child_gfs);
```

In this example, we have created a composite space that contains N children of type `ChildGFS`.

4.3.1 Multi Domain Function Space

The composite function spaces that are part of the standard PDELab framework do not support multi domain problems; all leaf spaces in a function space tree must be defined on the same mesh (represented by a DUNE `GridView`).

We thus need a new composite space implementation that can combine subspaces on different subdomains. In our implementation, this multi domain support is built on top of Dune-Multidomaingrid: A multi domain function space works in collaboration with an underlying `MultiDomainGrid` to track the domains of its child spaces. Each of those child spaces has to be a regular PDELab function space (it is allowed to be a composite space itself) and must be defined on a `GridView` of either the `MultiDomainGrid` or one of its `SubDomainGrid`s. Note that it is not possible to combine arbitrary grid views: for example, if one subspace uses the leaf grid view of its subdomain, all other spaces must also be defined on the leaf grid view of their respective subdomains.

In mathematical notation, we can describe a multi domain function space by

Definition 4.1 (Multi domain function space)**.** *Let $V = V_0 \times \cdots \times V_{n-1}$ a function space formed by the tensor product of elementary function spaces V_i of functions $v_i : \Omega_i \to \mathbb{R}^d$ with $\Omega_i \subset \mathbb{R}^d$ a subset of the overall simulation domain $\Omega = \Omega_0 \cap \cdots \cap \Omega_{n-1}$.*

Given a sequence $(\mathcal{T}_0, \dots, \mathcal{T}_{n-1})$ *of meshes for the domains* $(\Omega_0, \dots, \Omega_{n-1})$ *and corresponding function spaces*

$$V_i^h : \{v \in V_i : v|_T \in \mathcal{F}_i \ \forall\ T \in \mathcal{T}_i\}, \quad i = 0, \dots, n-1,$$

the composite function space V^h *is defined by*

$$V^h = V_0^h \times \cdots \times V_{n-1}^h,$$

and for brevity we introduce the notation $\mathfrak{T} = (\mathcal{T}_0, \dots, \mathcal{T}_{n-1})$ *for the sequence of meshes supporting* V^h.

Translated into C++, this definition maps to a `MultiDomainGridFunctionSpace` object, which closely resembles the interface of the `CompositeGridFunctionSpace` shown above, but also requires the `MultiDomainGrid` to manage the subdomain relationships. As an example, consider the two-domain Poisson example (4.2): Given a pair of scalar function space objects `LeftGFS` and `RightGFS` for the left and the right part of the domain, the corresponding multi domain function space is constructed by

```
typedef MultiDomain::MultiDomainGridFunctionSpace<
  MultiDomainGrid,          // underlying mesh with subdomain information
  VectorBackend,
  LexicographicOrderingTag, // only lexicographic ordering supported
  LeftGFS,
  RightGFS
  > MDGFS;
MDGFS md_gfs(mdgrid,left_gfs,right_gfs);
```

4.3.2 Subproblem Subspaces

Due to the very nature of multi physics problems, their assembly requires us to operate on *subspaces* of the overall ansatz and test function spaces. Even when considering the basic example of two coupled Poisson problems, there are separate residuals for each subproblem, and each subproblem only assembles a standard Poisson residual on the scalar subspace for the associated subdomain Ω_L or Ω_R, respectively. We thus need a mechanism to designate and construct those subproblem subspaces from the global spaces. Within our framework, we support subspaces that consist of a subset of the direct children of the global `MultiDomainGridFunctionSpace`. In order to select those children, we introduce an *index tuple* as a sequence of indices $\mathcal{I} = (i_0, \dots, i_{N-1}), i_k \neq i_l\ \forall\ k \neq l$, where each index $i_k \in 0, \dots, (N-1)$ denotes a direct child node of the global multi domain space V.

Definition 4.2 (Subproblem Subspace)**.** *Given an index tuple* $\mathcal{I} = (i_0, \dots, i_{N-1})$ *specifying a set of child spaces and their ordering, we define the subproblem subspace* $V_\mathcal{I}$ *by*

$$V_\mathcal{I} = V_{i_0} \times \cdots \times V_{i_{N-1}}.$$

Note that by using a tuple, which preserves the order of the child indices, we can reorder the child spaces with respect to their canonical order in the global function space tree, which might be necessary if we want to incorporate two existing residual operators that assume a different order for their variables, as explained in the Stokes-Darcy example.

In general, the components of a subspace are not necessarily all defined on the same subdomain; if we want to evaluate a function defined on the subspace, we can only do so on the intersection of the domains of all components. This limitation is made explicit by the *restricted subspace*, which reduces the spatial domain of a normal subspace to this intersection:

Definition 4.3 (Restricted subspace)**.** *Let $V_{\mathcal{I}}$ a subspace of the global space V with $\mathcal{I} = (i_0, \dots, i_{n-1})$ and per-component meshes $\mathcal{T}_{V_{i_k}}$. Then the restricted mesh $\mathcal{T}_{\mathcal{I}}$ is given by*

$$\mathcal{T}_{\mathcal{I}} = \mathcal{T}_{V_{i_0}} \cap \cdots \cap \mathcal{T}_{V_{i_{n-1}}}$$

and the restricted subspace *$V_{\mathcal{I},\mathcal{T}_{\mathcal{I}}}$ induced by $V_{\mathcal{I}}$ is defined by*

$$V_{\mathcal{I},\mathcal{T}_{\mathcal{I}}} = V_{i_0}\big|_{\mathcal{T}_{\mathcal{I}}} \times \cdots \times V_{i_{n-1}}\big|_{\mathcal{T}_{\mathcal{I}}}.$$

There is no equivalent to these globally defined subspaces at the implementation level of Dune-Multidomain, as the subspaces described here are only required during problem assembly; in that context it is sufficient to provide a local function space as introduced in Section 2.2.3, which will be automatically synthesized by the software framework when required in the context of subproblem assembly (see next section). The technical details of this process are explained in Chapter 7.

4.4 Decomposition of Residuals Into Semantic Building Blocks

As shown in the examples at the beginning of this chapter the residual form $\mathcal{R}$ of a multi physics problem can be broken down into a sum of separate forms that correspond to individual physical models and often depend on only a subset of the problem variables. In the following, we will identify and define two particular types of these residual components which we call *subproblems* and *couplings* and which broadly correspond to single physics problems and pairwise interactions between those single physics models.

4.4.1 Subproblems: Single Physics Components

Almost any multi physics model starts with a number of established single physics models for the individual physical phenomena that are part of the overall problem, which are then coupled by incorporating additional multi physics interactions. In

the context of our framework, these building blocks are called subproblems; for many multi domain problems, they will not be implemented from scratch, but taken from an existing simulation for the single physics problem.

In order to specify a residual $\mathcal{R}_P$ for such a subproblem we obviously need the weak residual form r_P and its associated ansatz and test spaces, but this leaves the question of the spatial domain (essentially, the set of grid cells) over which r_{SP} should be integrated. The largest possible integration domain is that of the restricted subspace for the variables that occur in r_P. However, if we consider our two-phase flow example, we have the simplified single-flow model that is defined in terms of the water pressure. That water pressure is defined on the whole simulation domain, but at the same time, the single-phase flow residual must only be assembled in those parts where there is *no* variable for the oil concentration. We thus need a mechanism to explicitly select whether a subproblem residual should be assembled on a given grid cell. For this purpose, we introduce a mesh predicate $q : \mathcal{T} \to \{0, 1\}$. In the context of our framework, this predicate operates not on the grid entity of the cell itself, but on its subdomain set (cf. Section 3.1.3. For convenience, our framework includes two default predicate implementations:

```
MultiDomain::SubDomainEqualityCondition<
  Grid
  > equality_condition {2,7,3}; // cell must exactly belong to set {2,7,3}
MultiDomain::SubDomainSubsetCondition<
  Grid
  > subset_condition {2,7,3};   // cell must at least belong to set {2,7,3}
```

As their names imply, when created with a subdomain set s_P these predicates match a cell with subdomain set s if $s_P = s$ or $s_p \subseteq s$, respectively.

With the necessary components in hand, we can formally define a subproblem by

Definition 4.4 (Subproblem)**.** *Let $U_\mathcal{I}$ and $V_\mathcal{I}$ a pair of restricted subspaces of the global ansatz and test spaces U and V induced by the index tuple $\mathcal{I}$ on the meshes $\mathfrak{T} = (\mathcal{T}_1, \ldots, \mathcal{T}_n)$. Furthermore, let $q_P : \mathcal{T}_\mathcal{I} \to \{0, 1\}$ a predicate on the overlapping mesh $\mathcal{T}_\mathcal{I}$ and $\mathcal{T}_P = \{T \in \mathcal{T}_\mathcal{I} : q_P(T) = 1\}$ the subset of $\mathcal{T}_\mathcal{I}$ selected by that predicate. Finally, let*

$$\mathcal{R}_P : U_\mathcal{I}\big|_{\mathcal{T}_P} \to V_\mathcal{I}\big|_{\mathcal{T}_P}$$

a discrete residual form on the given subspaces. Then the supblem P is defined by the tuple

$$P = (U, V, \mathcal{R}_P, q_P, \mathcal{I}_P). \tag{4.13}$$

Within our framework, this mathematical definition directly translates to a corresponding `SubProblem` object. Considering the two-phase flow subproblem of (4.9), such an object can be created like this:

```
typedef MultiDomain::SubProblem<
  MultiDomainGFS,        // overall ansatz space
  MultiDomainGFS,        // overall test space
```

```
  TwoPhaseFlowOperator,  // local operator for residual
  Predicate,             // mesh predicate
  0,1                    // subspace index tuple I
  > SubProblem;
SubProblem sub_problem(
  two_phase_flow_operator,
  predicate
  );
```

A subproblem object does not support any operations apart from extracting the stored information; its only purpose is to encapsulate all information required to assemble the subproblem residual $\mathcal{R}_P$.

4.4.2 Coupling Nonoverlapping Subproblems

While subproblems as defined in Section 4.4.1 are capable of encapsulating the single physics building blocks of a multi physics problem and can even be used to model certain classes of couplings between those building blocks, they cannot represent residual forms that model surface interactions between nonoverlapping subproblems. From the point of view of form assembly, these surface couplings instead bear a close resemblance to the interior surface integrals of DG schemes (called skeleton integrals in PDELAB): They are assembled on a set of cell intersections, and there are separate local function spaces for the two adjacent cells. The main difference to a regular skeleton integral lies in the fact that for a skeleton integral, those two local spaces belong to the same global function space, while in the case of a coupling integral, the function spaces on each side of the coupling interface are usually not related. This makes it impossible to implement this type of integral in a standard skeleton method of a local operator. Instead, we have added a slightly extended `alpha_coupling()` method to the `LocalOperator` interface:

```
template<
  typename IG,
  typename SP1LFSU, typename SP1LFSV,
  typename SP2LFSU, typename SP2LFSV,
  typename X, typename R>
void alpha_coupling(
  const IG& cell_intersection,
  const SP1LFSU& sp1_lfsu, const X& sp1_x, const SP1LFSV& sp1_lfsv,
  const SP2LFSU& sp2_lfsu, const X& sp2_x, const SP2LFSV& sp2_lfsv,
  R& sp1_r, R& sp2_r
  ) const;
```

Its signature is virtually identical to the existing `alpha_skeleton()` method with separate function spaces, solutions and residuals for the grid cells on either side of `cell_intersection`. It only differs by allowing different C++ types for the spaces on each side. By convention, the first set of parameters (labeled `sp1_`) is associated with the inside cell of the intersection as defined by the DUNE grid interface, while the second set of parameters lives on the outside cell.

In keeping with the standard `LocalOperator` API, assembly of this term has to be enabled by a compile time switch:

```
class CouplingLocalOperator
  : CouplingOperatorDefaultFlags // turn off all assembly kernels by default
{
  // enable alpha_coupling() kernel
  static const bool doAlphaCoupling = true;

  ...
};
```

There are additional methods for matrix pattern and Jacobian assembly etc. that mirror the functionality of the stock PDELab interface, but which we omit here for brevity. A full example of a `Localoperator` class implementing this interface for the coupling residual of the Stokes-Darcy problem can be found in Listing A.2.

Given a coupling operator, we still need to define its integration domain and the spaces on each side of that domain. For this purpose, we introduce a second class of residual components called *couplings*, which represent the pairwise interaction between two subproblems and their associated variables on a codimension 1 manifold (usually the coupling interface):

Definition 4.5 (Coupling)**.** *Let* $P_1 = (U, V, \mathcal{I}_1, \mathcal{R}_1, \mathcal{T}_1)$ *and* $P_2 = (U, V, \mathcal{I}_2, \mathcal{R}_2, \mathcal{T}_2)$ *two subproblems with non-overlapping meshes* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *that touch along a coupling interface* $\mathcal{T}_C = \mathcal{T}_1 \cap \mathcal{T}_2 \neq \emptyset$*. Moreover, let the sets of variables in* P_1 *and* P_2 *be disjoint, i.e.* $\{i \in \mathcal{I}_1\} \cap \{i \in \mathcal{I}_2\} = \emptyset$*. Furthermore, let*

$$r_C : \left(U_{\mathcal{I}_1} \times U_{\mathcal{I}_2}\right)\Big|_{\mathcal{T}_C} \times \left(V_{\mathcal{I}_1} \times V_{\mathcal{I}_2}\right)\Big|_{\mathcal{T}_C} \to \mathbb{R}$$

a residual form on the coupling interface $\mathcal{T}_C$ *describing the coupling mechanism between the two subproblems* P_1 *and* P_2*. Then the coupling* C *is defined by the triple*

$$C = (P_1, P_2, \mathcal{R}_C).$$

Note that the order of the subproblems in the definition of the coupling has to be identical to the argument order of r_C.

This definition again maps to C++ in a very straightforward fashion: In our framework, a coupling as defined above is represented by a `Coupling` object that captures the two subproblems and the local operator for the coupling residual r_C. Considering the coupled Poisson example, this object is created by

```
typedef MultiDomain::Coupling<
  LeftSubProblem,
  RightSubproblem,
  CouplingOperator
  > Coupling;
Coupling coupling(
  left_sub_problem,
```

```
  right_sub_problem,
  coupling_operator
  );
```

Given this `Coupling` object, our framework will make sure to always invoke the callback methods of the coupling local operator in such a way that the function space corresponding to the left subproblem is passed as the first argument and that the normal of the intersection points from the left to the right subproblem.

4.4.3 Constraints Handling

When implementing a standard FEM problem, constraints are mostly limited to DOFs on the outer domain boundary and internal process boundaries, which both coincide with the outer boundaries of the (process-local part of the) grid.

This assumption does not hold for multi domain problems, where multiple internal subdomain boundaries can occur; moreover, the constraints of those boundaries have to be applied only to their associated subproblem subspaces and their DOFs.

In order to assemble the constraints, our multi domain framework contains a more capable replacement for the default PDELab constraints assembler. As explained in Section 2.3.3, the default assembler iterates over the entire grid and invokes assembly methods on the constraints engines of the leaf spaces (there are different methods for constraints e.g. on the domain boundary or on internal processor boundaries). In addition to the function space itself, the assembler also accepts a tree of parameter objects to e.g. specify the boundary condition type (Dirichlet or Neumann) at a given position

We have seen earlier that most standard PDELab operations can be mapped to the multi domain framework by applying them to subproblems instead of the entire domain and function space. This also holds for the constraints assembly: We typically want to add constraints to the border of a subproblem domain, as it corresponds to the border of its residual form. A typical multi domain problem will contain several subproblems which we might all want to constrain. For this purpose, the framework constructs a problem-specific constraints assembler that knows what constraints to apply to each subproblem. In the case of our Stokes-Darcy problem, it is constructed by

```
auto constraints_assembler = MultiDomain::constraints<RF>(
  multi_gfs,
  MultiDomain::constrainSubProblem(
    stokes_sub_problem,
    stokes_boundary_type
  ),
  MultiDomain::constrainSubProblem(
    darcy_sub_problem,
    darcy_boundary_type
  )
);
```

In standard PDELab, this assembler is automatically created when the user calls `PDELab::constraints()` and destroyed when that call finishes. In our case, this object will in general be much more complex (and expensive to build), so we have made its creation a separate, user-controllable step. Note that the exact type of the assembler is an implementation detail and depends on the nested calls to `constrainSubProblem`; users should always capture it in an **`auto`** variable. Once the constraints assembler has been created, we can call it to create the constraints map for `multi_gfs`:

```
typename MultiGFS::
  template ConstraintsContainer<RF>::Type cg; // create constraints container
constraints.assemble(cg);                     // start constraints assembly
```

4.4.4 Interpolation

When interpolating in a multi domain setting, we are again faced with the problem that we cannot specify a single, vector-valued function with values for all variables and perform a global interpolation.

As before, we solve this problem by means of the subproblems. Instead of interpolating a single, global function across the whole subdomain into the entire multi domain function space, we provide a list of subproblems and functions for the variables in each subproblem. For example, if the initial solution for the coupled Poisson problem is given by the scalar functions g_L and g_R, we can interpolate them by calling

```
MultiDomain::interpolateOnTrialSpace(
  md_gfs,
  u,
  g_left,left_subproblem,
  g_right,right_subproblem
  );
```

Note that we have to specify whether we are interpolating on the trial or the test space; as a subproblem stores subspaces for both spaces, the framework by itself cannot know which one of those to pick.

4.4.5 Assembly

After defining the building blocks of a multi domain problem, we now want to use these components to actually solve the corresponding algebraic system. For now, we concentrate on the assembly part.

PDELab encapsulates this global assembly in a `GridOperator`, which implements the generic FEM assembly as specified in Algorithm 2.1 and is parameterized on the ansatz and test function spaces as well as the cell- and intersection-local residual. The default implementation, which is part of PDELab, only supports function spaces that are defined on a single mesh as well as a single residual form (cf. Section 2.3.3).

In our multi domain setting, the assembly process and thus the grid operator is more complicated. In addition to the default version, it must be able to (1) cope with subspaces that are only defined on parts of the mesh, (2) manage a list of multiple subproblems and couplings and figure out on which parts of the global mesh they need to be assembled, and (3) prepare the restricted subspaces as specified in each subproblem / coupling.

Despite all those changes, the interface of our multi domain grid operator is almost identical to the stock version; the only user-visible change is that the single local operator passed to the standard operator is replaced by a list of subdomains and couplings. For example, the grid operator for the Poisson-Poisson example is created by

```
typedef MultiDomain::GridOperator<
  MDGFS,MDGFS,
  ISTLMatrixBackend,
  double,double,double,C,C,
  LeftSubProblem,
  RightSubProblem,
  Coupling
  > GridOperator;

GridOperator grid_operator(
  md_gfs,md_gfs,
  left_subproblem,
  right_subproblem,
  coupling);
```

Note that the operator will only work with `MultiDomainGridFunctionSpace`s; attempting to use it with a stock PDELab function space will cause a compilation error.

Once the operator has been created, it can be used in exactly the same way as the stock `GridOperator`; most importantly, it is compatible with the existing solver infrastructure. Consequently, we can use those existing components to solve the problem in a monolithic fashion by plugging it into one of the generic PDELab solver components, e.g. the default PDELab Newton solver:

```
typedef Newton<
  GridOperator,
  LinearSolverBackend,
  typename GridOperator::Domain
  > Solver;

Solver solver(
  grid_operator,
  linear_solver_backend,
  1e-10);

solver.apply(u);
```

On the other hand, by combining the multi domain grid operator with the flexible DOF mapping framework described in Chapter 6, we are able to efficiently create block matrices that can be used to implement loosely coupled solvers. An example of such a solver for the coupled Poisson problem can be found in Section 8.1.

While this chapter has only given a high-level overview of the interface to our multi domain framework, Chapter 7 describes some interesting parts of its implementation. However, before being able to dive into those details, the next two chapters introduce two essential building blocks of our software concerned with the handling of the function space trees and the construction of a global DOF ordering.

CHAPTER 5

Compile Time Polymorphic Trees and Associated Algorithms

As we have seen in the previous chapter, PDELab represents a composite function space as a tree with elementary function spaces as leaves. Moreover, most objects related to the function spaces are also implemented as trees (e.g. the `LocalFunctionSpace`s that represent restrictions to a single grid cell, analytic functions for initial values and the DOF mappings introduced in Chapter 6). Consequently, major parts of PDELab consist of operations on those trees. In the original PDELab implementation, all of those operations were implemented in an ad-hoc fashion wherever they occured in the code. This approach did not scale to the more complex tree structures that occur in our Dune-Multidomain extension module; we thus analyzed the exact requirements of our framework and created the dedicated TypeTree library that provides a generic implementation of those tree structures and the associated algorithms.

5.1 Introduction

TypeTree is a mostly freestanding library that can be used independently from DUNE. It is free software and available under the same licence as the DUNE core modules (the GNU General Public Licence with a special runtime exception, for further details see [40]). As its build system is currently built on top of the DUNE build system, it requires the Dune-Common module for building, but not at run time.

The following description is based on version 2.3.1 of the library, which can be downloaded from [97] or directly from the source code repository at [95]. It requires

the corresponding 2.3.1 release of the DUNE-COMMON module.

5.2 Problem Setting and Design Considerations

Before creating our own tree library, we started by looking at the requirements for such a library based on its application in PDELAB and extracted the following key properties:

- Trees in PDELAB are *structurally immutable*, i.e. it is not possible to add or remove children from an existing tree node.
- Consequently, these trees are always built in bottom-up fashion by starting with the leaf nodes, combining those into subtrees and then recursively continuing to aggregate those subtrees. A good example of this approach is the way function spaces are built up in PDELAB, which we have demonstrated in the previous chapter.
- The user payload inside the tree is heterogeneous; typically, each node stores user data with a unique C++ type.
- There is no fixed *degree* (number of children per node) for a given tree, so in addition to the heteregenous user data, the *structure* of the tree is also not uniform.
- Tree traversal is very performance-sensitive, as the framework needs to perform multiple traversals of different trees for each visited grid cell.
- In several places, PDELAB needs to algorithmically build new trees out of old trees (e.g. to construct a tree of cell-restricted function spaces from the tree of global spaces).

Trees have always been an important data structure in computer science [78], and so we initially tried to find an existing library for the trees in PDELAB. Surprisingly, there are only very few C++ libraries that provide a tree data structure, and those mostly implement some version of balanced binary trees for search applications [27, 71], while the few more general implementations like [103] are geared towards easy mutability and top-down construction and do not offer sufficient performance.

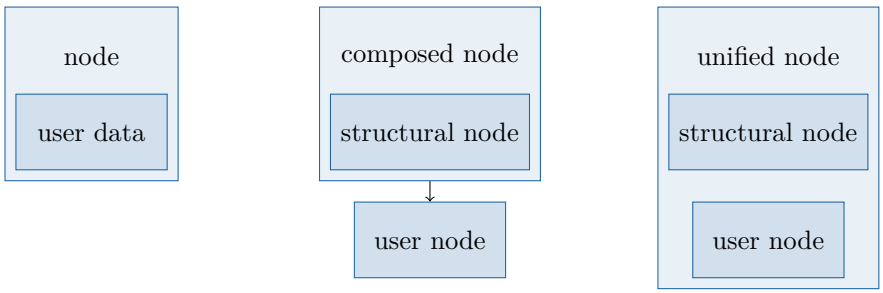
If we take a closer look at the properties listed above, they really boil down to the combination of two central requirements: (1) The heterogeneous nature of the trees, and at the same (2) the need for extremely fast grid traversal and access to node-specific behaviors and data during the traversal. The only way to reconcile those two requirements in C++ is by means of templates and static polymorphism: While the performance of dynamic polymorphism has been extensively optimized both at the compiler and at the hardware level [39, 28, 1, 106], the associated run time overhead still renders this approach unfeasible for our application. This is mostly due to the very fine granularity of many DUNE and PDELAB interfaces. Those functions often perform no more than a few machine instruction to calculate

	compiler		run time		overhead
	vendor	version	inlining t_i[s]	no inlining t_n[s]	t_n/t_i
Poisson Test	GCC	4.8.1	27.51	176.88	6.43
	clang	3.4pre	27.72	174.47	6.29
Instationary Advection	GCC	4.8.1	90.66	492.57	5.43
	clang	3.4pre	84.87	463.88	5.47
DNAPL FVM	GCC	4.8.1	129.07	989.83	7.67
	clang	3.4pre	122.87	953.66	7.76

Table 5.1 — Performance impact of function inlining in PDELab. Benchmark was performed with hardware configuration B.1. Inlining was disabled by appending `"-fno-inline"` to the standard optimization flags.

their result and thus rely on function inlining for good performance. Unfortunately, dynamic polymorphism presents an insurmountable barrier for function inlining, precisely because the decision about which code to execute is postponed until run time. Table 5.1 demonstrates the effect of enabling or disabling inlining for a number of real-world PDELab applications. As these numbers show, the typical performance advantage of inlining is a factor of ≈ 6.

While we were unable to find a statically typed, templated tree library for C++, simple heterogeneous containers have always been part of the language's standard library. The most prominent example is probably `std::pair`, which can store two unrelated objects; C++11 expands on this concept by providing tuples of arbitrary length. Outside of the standard library a much more extensive and very successful collection of heterogeneous general purpose containers and associated algorithms surfaced as part of the Boost framework [114] in the form of the Boost Fusion library: Guzman, Marsden, and Schwinger [62] pioneered methods for combining compile time TMP algorithms and containers from another well-known Boost library, Boost MPL [61, 3], with run time behavior modeled after the well known STL, part of the C++ standard [72]. Among others, Boost Fusion provides versions of the well-known container categories `vector`, `list` and `map` that support storing unrelated types in a single container. In several places, the internal design of our library is heavily inspired by techniques that were pioneered by Boost Fusion, in particular by its heavy use of tag dispatch to drive static polymorphism.



(a) Node with primitive data

(b) Inclusion of user behavior via composition

(c) Unified node with direct access to structural and user behavior

Figure 5.1 — Composition vs. inheritance for payload attachment to library data structures

5.3 Tree Nodes

In a recursive data structure like a tree or a linked list, the individual elements do not only contain the user data; they also have to store additional housekeeping information for the data structure, e.g. the link to the next element in a linked list. There are two fundamental approaches to this problem:

Intrusive containers store the additional information directly inside the user payload. In order to do so, the user data type has to be adapted to the container. In C++, this can e.g. be achieved by inheriting from a container-specific *mixin*:

```
1  // mixin class
2  template<typename Data>
3  struct ListNodeMixin
4  {
5    ListNodeMixin* next;  // pointer to next node
6  };
7  // user data
8  struct UserData
9    : public ListNodeMixin<UserData> // list data stored within user data
10 { ... };
```

It should be obvious that this technique is not well suited for general purpose containers: For every kind of container in which we want to store our data (list, set, map, ...), we have to inherit from a special mixin class. This is problematic if we cannot modify the user data (e.g. because it is from an external library).

On the other hand, if we know that our data fundamentally *is* a linked list, it can be very useful to make that data type aware of the list structure.

For example, many of the PDELab algorithms for function spaces have to traverse the function space tree. With an intrusive data structure, we can add methods to the user data type (the function space class) that performs those traversals.

Non-intrusive containers on the other hand store container data and user payload separately in a container-internal wrapper object. For a linked list, this might be implemented at the container level as

```
template<typename T>
struct Node
{
  T data;      // user data stored inside generic grid node
  Node* next;  // pointer to next node
};
```

Here, we have inverted the advantages and disadvantages of the other approach: The payload does not have to be aware of the container and we thus do not depend on help from the stored data. But now user code that works with such a list now has to be aware of the additional `Node` object. This approach works much better for a general purpose list library: The code that uses the list is naturally aware of the fact that the data structure is a list, while the data stored inside the list can stay oblivious to this fact.

Due to these trade-offs, most utility libraries like the C++ STL provide non-intrusive data structures. TypeTree does not follow this example; its trees are intrusive data structures. The trees that we build with the library (e.g. function spaces) fundamentally *are* trees, so we are not really concerned about generality; in our case, the benefits of the more integrated user-level API afforded by the intrusive approach are far more important.

TypeTree handles the structural information stored in the tree nodes (i.e. the lists of children) in a very flexible and extensible manner; as we will see later on, its design is based around algorithms that consist of isolated, node-specific building blocks which are looked up using a tag embedded in each node. Typically, when writing a *payload node* (e.g. a leaf or a composite function space), users select one of the predefined *structural node types*, which makes it possible to place the object in a TypeTree tree. The default nodes shipped with the library are implemented as mixin base classes that the user simply inherits from. They take care of storing the children of the node, provide an API for accessing those children and add the tag mentioned above. In the case of a leaf node (which does not need to store any children), this is very straightforward:

```
template<typename FEM>
class GridFunctionSpace
  : public TypeTree::LeafNode
{ ... };
```

The mixins for interior nodes are more interesting, as they need to contain information about their children, and are described in the following sections.

Listing 5.1 — VariadicCompositeNode interface

```
template<typename... Children>
class VariadicCompositeNode {
public:
  static const std::size_t CHILDREN = sizeof...(Children);

  template<std::size_t k>
  struct Child
  { /* access to static information about k-th child */ };

  template<std::size_t k>
  typename Child<k>::Type& child()
  { /* access to k-th child object */ }

  VariadicCompositeNode(Children&&... children)
  { /* construct from passed-in children */ }

  VariadicCompositeNode(std::shared_ptr<Children>... children)
  { /* construct from passed-in children */ }
};
```

5.3.1 VariadicCompositeNode

The default mixin for interior tree nodes, the `VariadicCompositeNode` is capable of managing interior nodes with an arbitrary number of heterogeneous children. Its usage resembles a C++ `std::tuple`: the types of the children are passed as a list of template parameters. For a very compressed overview of the API of the `VariadicCompositeNode`, see Listing 5.1).

The implementation relies on variadic templates and is thus only available on C++11 compliant compilers. Internally, the children are stored in a tuple of `std::shared_ptr`s; the constructor accepts either a list of `std::shared_ptr`s, which are simply copied into the internal tuple, or a list of references. In this case, we have no information about the ownership of those references and thus the children are not destroyed together with the `VariadicCompositeNode`. This way, users can create all tree nodes on the stack, which is a very common idiom in PDELab, e.g. when constructing a function space tree:

```
// leaf spaces created on the stack
GFS1 leaf_gfs_1(...);
GFS2 leaf_gfs_2(...);
typedef CompositeGridFunctionSpace<...,GFS1,GFS2> CompositeGFS;
// composite space now contains pointers to stack objects
CompositeGFS composite_gfs(leaf_gfs_1,leaf_gfs_2);
```

5.3.2 CompositeNode

The `CompositeNode` is designed as a fallback implementation of the default composite node type (`VariadicCompositeNode`) for compilers that lack variadic template

Listing 5.2 — CompositeNode fallback compatibility macros

```
template<typename T1, DUNE_TYPETREE_COMPOSITENODE_TEMPLATE_CHILDREN>
class CompositeUserData
  : public DUNE_TYPETREE_COMPOSITENODE_BASETYPE
{
  typedef DUNE_TYPETREE_COMPOSITENODE_BASETYPE NodeT;

public:
  typedef CompositeUserDataTag ImplementationTag;

  CompositeUserData(T foo, DUNE_TYPETREE_COMPOSITENODE_CONSTRUCTOR_SIGNATURE)
    : NodeT(DUNE_TYPETREE_COMPOSITENODE_CHILDVARIABLES)
  {}

  CompositeUserData(T foo, DUNE_TYPETREE_COMPOSITENODE_CONSTRUCTOR_STORAGE_SIGNATURE)
    : NodeT(DUNE_TYPETREE_COMPOSITENODE_CHILDVARIABLES)
  {}
};
```

support. At the time when TypeTree was initially incorporated into PDELab, there were are number of large machines without access to compilers with C++11 support (most notably JUGENE at Jülich Supercomputing Centre (JSC)). PDELab support for these machines was critical in the context of several research projects, so we created the alternative `CompositeNode`, which uses a fixed template parameter list; as a consequence, it is restricted to a maximum of 10 child nodes. If the user provides less than 10 children, the remaining template parameters default to a special marker type that denotes a missing child. Overall, this makes it possible to emulate most user-visible parts of the `VariadicCompositeNode`. Unfortunately, this does not hold for the payload node type sitting on top of the `CompositeNode`: While it doesn't use the template list of children internally, it has to duplicate it, so in a naive implementation, it would be necessary to provide alternative versions of all composite payload types (grid function spaces, local function spaces, grid functions, ...) in the library. This seemed excessive, so we developed a workaround for this problem in the form of a mostly transparent mechanism to automatically replace `VariadicCompositeNode` with `CompositeNode` throughout the PDELab code base. It relies on a number of preprocessor macros to insert the correct code into the definition of the payload data structure; as can be seen in Listing 5.2, this mostly involves changes to the template signature of the new class, the type of the tree node mixin and the argument signatures of the constructors that forward the child nodes to the mixin class. TypeTree will then automatically detect whether the compiler supports variadic templates and select the appropriate definitions for those macros to use either `VariadicCompositeNode` or `CompositeNode`.

Before the integration of TypeTree into PDELab, its composite function space always used an emulation for the variadic template arguments; switching to "real" variadic template arguments yielded a massive improvement in compilation times

(up to a factor of 2) for modern compilers. As compile times are rather long for template-heavy code like DUNE, this improvement markedly increased the developer usability of the framework.

5.3.3 PowerNode

In theory, the composite node described above is sufficient to be able to map arbitrary trees into the TypeTree framework, but for usability and performance reasons, a specialized interior node containing only children of identical type has been part of PDELab since its inception. The `PowerNode` is specified by the type of its children and their number, which e.g. simplifies the construction of vector spaces from scalar spaces in dimension independent code. Moreover, for large number of children the type signature of a `PowerNode` will be far shorter than the one of an equivalent `VariadicCompositeNode`, improving compile times and readability of compiler error messages. Finally, it is possible to exploit the fact that the children of a `PowerNode` share a common type by switching loops over those children from compile time constructs to regular run time loops, further reducing the compiler burden (as the loop boundaries are still known at compile time, this does not preclude extensive optimization and loop unrolling by the compiler).

```
1  template<typename Child_, std::size_t N>
2  class PowerNode {
3  public:
4
5    // basic interface identical to VariadicCompositeNode
6
7    typedef Child_ ChildType; // access to unique child type
8
9    ChildType child(std::size_t)
10   { /* child access with run time indexing */ }
11
12   VariadicCompositeNode(ChildType&& c_1, ..., ChildType&& c_N)
13   { /* construct from passed-in children */ }
14
15   VariadicCompositeNode(std::array<std::shared_ptr<ChildType> >&& children)
16   { /* construct from array of children  */ }
17 };
```

5.3.4 Classifying Tree Nodes

As the list of children is encoded into the type signature of each TypeTree node, it is clear that all trees created with the library will be heterogeneous data structures; any algorithms that operate on those trees will thus have to be written as TMPs. This is true for any kind of heterogeneous data structure in C++, and the standard technique for performing an operation to each item in a heterogeneous data container consists of encapsulating the nodal operation in a generic functor

that is then applied to the container by means of a generic traversal algorithm. For example, in order to output each element of a `std::tuple` to the console, we can write the functor

```
struct Writer
{
  template<typename T>
  void operator()(const T& t)
  {
    std::cout << t << std::endl;
  }
};
```

and implement a tuple-specific iteration algorithm as a pair of overloaded template functions:

```
template<typename T, typename F>
void tuple_for_each(T& tuple, Functor f, std::integral_constant<int,0>)
{
  f(std::get<0>(t));
}

template<typename T, typename F, int i = std::tuple_size<T>::value - 1>
void tuple_for_each(T& tuple, Functor f, std::integral_constant<int,i> = {})
{
  tuple_for_each<T,F,i-1>(tuple,f);
  f(std::get<i>(t));
}
```

As it is not possible to write template code in iterative fashion, this is a recursive algorithm that uses the integral template parameter `i` to control the iteration. Given a tuple variable `my_tuple`, it can be invoked by

```
tuple_for_each(my_tuple,Writer());
```

This is a well-known idiom for user of the C++ STL, which contains a library of algorithms built on top of this call back pattern. The Boost Fusion [62] library also follows this approach and contains a large number of STL-like algorithms for its heterogeneous containers.

Writing a generic algorithm like the `tuple_for_each()` function above for TypeTree faces an additional challenge: Heterogeneous containers like `std::tuple` or `fusion::map` store heterogeneous user data, but there is only a single (templated) implementation of the container nodes (cf. the list example in Section 5.3). In the TypeTree library, the situation is more complicated because there are multiple node implementations which all require different code to iterate over their children. Consequently, the TypeTree algorithms must be able to differentiate between those node types. This is typically done either by (partially) specialization of the algorithm components (e.g. the `for_each` above) for every possible type of node or by using some form of tag dispatch, as described in Section 2.4.1. Specialization does not work in this case because the mixin type is hidden in the list of base

classes, so TypeTree uses tag dispatch. For simplicity, every node mixin explicitly exports its *node tag* via a public `typedef` called `NodeTag`.

Some algorithms in TypeTree (tree transformations in particular) are additionally controlled by the *user type* of a node. For consistency and to avoid the fragility of partial template specialization, user types are also classified using a second tag type, which the user has to export under the name `ImplementationTag`. Taken together, these two tags make it possible to have completely different user payloads that share a common TypeTree node type (for example, PDELab has a `PowerGridFunctionSpace` and a `VectorGridFunctionSpace`, both implemented as a `PowerNode`) as well as the reverse (the same type of user node sitting on top of different tree topologies – this makes it very easy to implement *proxy nodes*, a feature that is heavily used by Dune-Multidomain for its subproblem function spaces).

5.4 Algorithms

In addition to the node implementations, TypeTree also contains a number of algorithms that work on those trees. In the following, we present the two most important of those algorithms: tree traversal and tree transformations.

All of the algorithms in TypeTree work by dispatching to specialized, node-local building blocks based on these two tags. The resulting algorithms are highly modular, which ensures good maintainability. More importantly, the tag dispatch mechanism also causes them to be coupled in an extremely weak fashion, which makes it very easy to extend them to work with a new node type – after writing the required algorithm pieces and hooking them up to the dispatch mechanism, the existing parts of the implementation do not need to be modified in any way to incorporate the new node type.

5.4.1 Tree Traversal

Due to the hierarchical structure of a tree, there is no single canonical traversal order for its nodes. In general, computer science distinguishes two fundamental traversal algorithms:

Breadth-first traversal involves visiting the nodes of the tree in top-down order. It starts at the root node, goes on to visit all direct children of the root node and then iteratively continues with the i-th level descendants of the root node, in that order. It is usually implemented by means of a *queue*: Every time the algorithm encounters a new node, its children are pushed onto the end of the queue, and the algorithm advances by popping nodes off of the front of the queue until exhaustion.

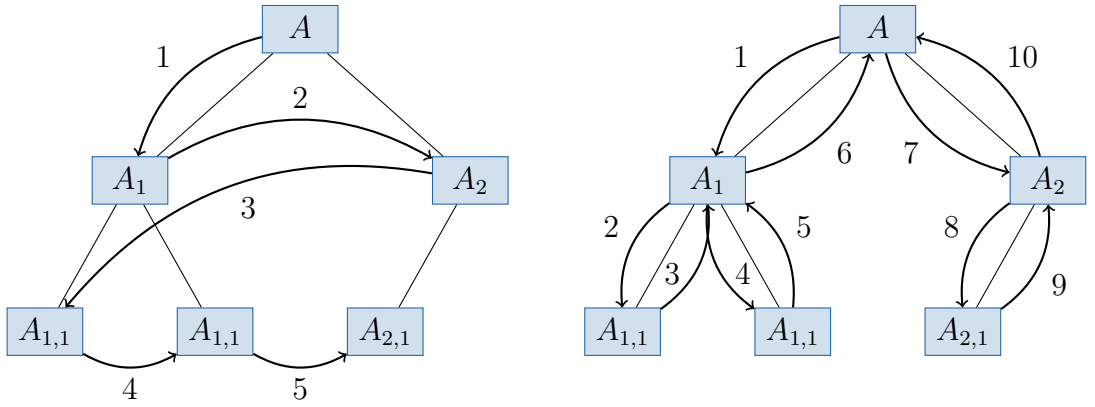


Figure 5.2 — Breadth-first and depth-first tree traversal

Depth-first traversal does instead follow a given path down the tree as far as possible and then continues to search for the next eligible path by means of backtracking. This corresponds to using a *stack*, where elements are both added to and retrieved from the same end of the data structure.

Figure 5.2 shows the traversal path through an example tree for both breadth-first and depth-first traversal. Interestingly, implementing the latter algorithm for our trees is relatively straightforward, while a breadth-first traversal would be very difficult to realize. This is due to the fact that the traversal algorithm has to be written as a TMP, and the stack-based nature of the depth-first traversal naturally matches the recursive programming style of template meta programming; we can simply use the call stack of our program to store the algorithm state. On the other hand, manually implementing an efficient queue for heterogeneous data using C++ templates seems like a very hard task.

Fortunately, the intended applications in our framework only require depth-first traversal, so we have restricted ourselves to that kind of iteration. If we take a closer look at the algorithm, it becomes clear that due to its backtracking property, the traversal trajectory passes interior tree nodes multiple times. Depending on when the algorithm stops at those nodes, depth-first traversal can be further categorized:

Pre-order traversal visits interior nodes as soon as they are encountered for the first time. It is useful for algorithms that need to propagate information down the tree.

In-order traversal is mostly important for ordered binary trees, where it visits the nodes in the sorting order.

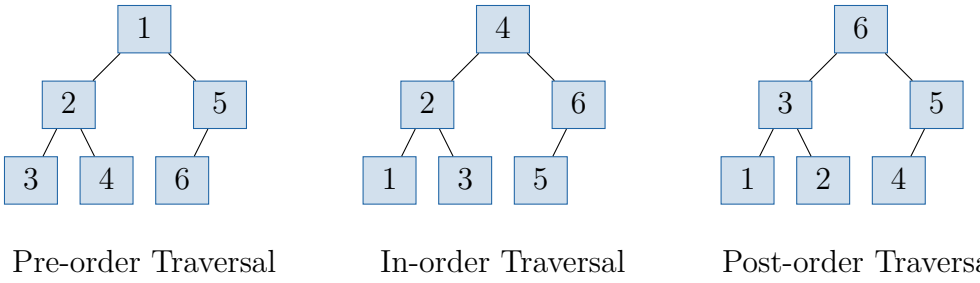


Figure 5.3 — Pre-, in- and post-order depth-first tree traversal

Post-order traversal must be used when operations on interior nodes require the results of all descendants, i.e. information has to be moved up the tree.

Figure 5.3 depicts the order in which the nodes of an example tree are visited for each of the three variants.

In order to separate the iteration algorithm from the operations performed on the tree nodes, computer science has developed the *visitor pattern* [99, 26]: a generic algorithm traverses the tree and presents individual elements to the visitor by invoking its callback function. In the context of trees, a visitor may have multiple callback functions that are called at different times to implement the different traversal orders. This allows a single visitor to e.g. perform a combined pre- and post-order traversal.. Listing 5.3 shows the callback points offered by the visitor interface of TypeTree. In addition to the actual node, the callbacks also receive information about the position of the node within the tree via the additional `TreePath` parameter, which encodes the path from the root to the current node as a tuple of child indices. While the `pre`, `in` and `post` callbacks are commonly encountered in tree libraries, our interface also contains two additional methods that help in writing algorithms where data has to be moved up or down the tree hierarchy. The design of the data structure makes this difficult to accomplish manually because there is no link from child to parent. Finally, the visitor also contains a template meta function that is called for each node to decide whether or not traversal should continue into the children of that node (in the example above, that function is inherited from the base class). This is an important optimization, as the tree traversal is a TMP itself and thus completely unrolled; unnecessarily traversing a large number of deep tree hierarchies would greatly increase compile times of TypeTree-based programs.

The actual traveral Algorithm 5.1 is based on a modular framework based on compile time dispatch to node-specific iteration logic. Depending on the characteristics of the current node, this logic may employ run time iteration (which has the advantage of reducing code size and compile time, but is only possible for homogeneously structured nodes like `PowerNode`) or compile time recursion. At the same time, this modular approaches ensures easy extensibility, as new node types can be

Listing 5.3 — TypeTree visitor interface

```
struct Visitor
  : public TypeTree::DefaultVisitor
{
  // pre-order callback, called for interior nodes
  template<typename Node, typename TreePath>
  void pre(Node&& node, TreePath tree_path)
  {}

  // in-order callback, called for interior nodes
  template<typename Node, typename TreePath>
  void in(Node&& node, TreePath tree_path)
  {}

  // post-order callback, called for interior nodes
  template<typename Node, typename TreePath>
  void post(Node&& node, TreePath tree_path)
  {}

  // callback for leaf functions
  template<typename Node, typename TreePath>
  void leaf(Node&& node, TreePath tree_path)
  {}

  // called before traversing down a father-child relationship with information
  // about both father and child; simplifies data propagation down the tree
  template<typename Node, typename Child, typename TreePath, typename ChildIdx>
  void beforeChild(Node&& node, Child&& child, TreePath tp, ChildIdx ci)
  {}

  // called before traversing up a child-father relationship with information
  // about both father and child; simplifies data propagation up the tree
  template<typename Node, typename Child, typename TreePath, typename ChildIdx>
  void afterChild(Node&& node, Child&& child, TreePath tp, ChildIdx ci)
  {}
};
```

accommodated by adding a new dispatch overload. Importantly, adding this overload does not require any changes to existing algorithm components.

5.4.2 Simultaneous Traversal of Tree Pairs

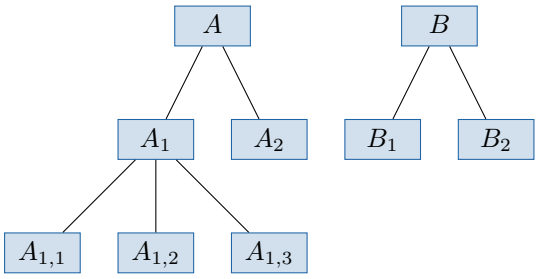
Within the context of PDELab, there is a recurring need to traverse a tree and apply a function that needs data which is stored in a different tree, e.g. the data required to interpolate a function or evaluate the constraints (cf. Section 2.3.3). Essentially, we need to traverse two trees in parallel and present the visitor with matching pairs of tree nodes. TypeTree contains an extended iteration algorithm that enables this usage scenario.

Algorithm 5.1 — Tree traversal algorithm. N is the current tree node, V the visitor, and p the tree path. In this algorithm, we show a single `apply()` function with its general semantics. The dispatch stage uses the component registry $\mathcal{D}$ to look up a version of `apply()` that is tailored to the node type of the current node N and implements the apply functionality in a way that is optimized for the data layout of the node.

```
function DISPATCH(N, V, p)
    f ← D[TAG(N)]
    f(N, V, p)

function APPLY(N, V, p = ())
    if V.WANTSTOVISIT(N) then
        if N is leaf node then
            V.LEAF(N, p)
        else
            V.PRE(N, p)
            n ← |CHILDREN(N)|
            for i ← 1, n do
                C ← CHILDREN(N)[i]
                q ← p ‖ (i)
                V.BEFORECHILD(N, C, p, i)
                DISPATCH(C, V, q)          ▷ dispatch to apply function for child
                V.AFTERCHILD(N, C, p, i)
                if i < n then
                    V.IN(N, p)
            V.POST(N, p)
```

In general, it is desirable not to require the two trees to be structured identically, as long as they are compatible in the sense that corresponding nodes in the two trees have either an identical number of children, or at least one of the nodes does not have any children at all. With this relaxed requirement, it becomes possible to iterate over tree pairs where the two trees employ different node types for identically positioned nodes and to support scenarios where one tree is cut off at some interior nodes. If we encounter such a cut-off scenario, it is no longer obvious how the traversal should proceed: We can continue traversing the "deeper" tree and call the visitor with all resulting combinations of the additional nodes in that tree and the single node in the other tree, which leaves us with the question of how to distinguish this special case in the visitor. For that reason, we have instead opted to simply ignore the additional tree nodes in the deeper tree and to invoke the leaf callback on the visitor with the two corresponding nodes that introduced the discrepancy. If a visitor needs to continue the traversal, it is trivial



(a) Non-conforming trees

Node 1	Node 2	Callback
A	B	pre
A_1	B_1	leaf
A	B	in
A_2	B_2	leaf
A	B	post

(b) Visited node pairs

Figure 5.4 — Simultaneous traversal of non-conforming trees

to set up a standard single-tree traversal starting at the passed-in node. Figure 5.4 demonstrates the semantics of the simultaneous tree traversal.

5.5 Tree Transformations

The design of TypeTree is inspired by functional programming, which generallly regards data as immutable and expresses programs in terms of functions that produce new data from a given input [70]. This paradigm manifests itself in the "tree" portion of TypeTree: Once a node has been created, it becomes impossible to add or remove children from it (the default node implementations do actually allow replacing a child with another object of the same type, but that is mostly an implementation detail and not used in any of our trees). On the other hand, the data contained within the node, which imbues the tree with actual meaning, may be changed at any time.

In the context of PDELab, this split view (immutable tree structure, mutable payload data) fits very well: While the data stored in a function space tree is very much expected to change (possible reasons include grid refinement, p-adaptivity or load balancing for parallel computations), we typically don't want to add or remove new variables after creating the space, so its *structure* is frozen after creating the tree.

On the other hand, PDELab defines a number of data structures which depend on the function space and that are stored in trees with identical (or at least similar) shape. Early versions of PDELab either constructed these dependent trees using one-off special purpose code (e.g. to construct the `LocalFunctionSpace` tree of cell-restricted spaces) or required the user to build the new tree manually. As TypeTree trees are constructed in a bottom-up fashion, this process typically involves completely deconstructing the old tree to its leaves, defining new leaf nodes based on those old leaves and then recursively combining the new leaves to create

the new tree, a very verbose and error-prone procedure if performed manually by the user.

Most of the time in PDELab, this ad-hoc procedure actually follows a general algorithm that takes an input tree and, according to a set of node-specific rules, returns a new tree. In the following, we call such an algorithm a *transformation* $\mathcal{F}$. In mathematical terms, a transformation $\mathcal{F} : \mathcal{B} \to \mathcal{B}'$ is a mapping from one set of trees $\mathcal{B}$ to another $\mathcal{B}'$. This definition allows for a broad range of transformations, including returning a tree with a different shape than the original, a feature that will be important in some of our applications.

Looking at the majority of transformations in PDELab, the resulting tree is identical (or at least very similar) in shape to the input tree. We have found that this type of transformation can often be written as a combination of two components:

- A generic algorithm that is only concerned with a depth-first traversal of the tree and producing the transformed tree nodes in post order (that way, children are transformed before their parents, as required to construct a TypeTree tree),
- and a set of transformation descriptors, one for each type of user data, that are used by that algorithm to convert each individual node.

This separation of concerns decouples the overall transformation process into two largely orthogonal parts and makes it easier to maintain and extend both of those parts: Users who want to define a new transformation only have to describe the result of the transformation for each of their user node types and do not have to contend with the way the underlying TypeTree nodes handles their children. At the same time, after adding a new structural TypeTree node type to the library, that type only has to be integrated into the generic traversal algorithm and will then work for all user-defined transformations implemented on top of that algorithm.

Performing a transformation is very straightforward and works by invoking the meta function `TypeTree::TransformTree` with the root of the start tree and a tag type that identifies the transformation. For example, the transformation that takes a function space tree and creates its corresponding ordering tree (cf. Chapter 6) is called `gfs_to_ordering`. Given a root space `RootGFS`, we can create and invoke it like this:

```
1 // Create the transformation
2 typedef TypeTree::TransformTree<
3   RootGFS,
4   gfs_to_ordering<RootGFS>
5   > Transformation;
6 // Compile time part: Generate type of resulting tree
7 typedef typename Transformation::Type OrderingTree;
```

Algorithm 5.2 — Tree transformation. Here, $\mathcal{F}$ denotes a transformation, $\mathcal{B}$ the source tree and d the user-specified transformation descriptor for the root node of $\mathcal{B}$ that can be found by a double dispatch on the transformation and the tree.

function APPLYTRANSFORMATION($\mathcal{F}, B$)
  $d \leftarrow$ DISPATCH($B, \mathcal{F}$)
  if d is recursive **then**
    $C \leftarrow$ CHILDREN(B)
    $n \leftarrow |C|$
    $C' = (\text{APPLYTRANSFORMATION}(\mathcal{F}, C[i]) : i \leftarrow 1, n)$
    $B' \leftarrow d(B, C')$
  else
    $B' \leftarrow d(B)$
  return B'

```
8 // Run time part: Create instance of transformed tree
9 OrderingTree ordering_tree(Transformation::transform(root_gfs));
```

The generic traversal as shown in Algorithm 5.2 is implemented in a similar fashion to the tree traversal in that it consists of a loosely coupled collection of building blocks, one for each type of TypeTree node, that are responsible for converting that single node and are looked up by dispatching on the TypeTree node tag. The algorithm does, however, incorporate additional user-provided components in the form of node-specific *transformation descriptors*. These are looked up by a double dispatch on the transformation tag and the `ImplementationTag` of the current node (cf. Section 5.3.4) and are responsible for transforming the user data inside the current node. Depending on their type, they can moreover control whether or not the generic algorithm will recurse into the children of the current node, a mechanism that will be described in detail in the next section.

5.5.1 Descriptor Structure

The generic transformation algorithm described in the previous section supports two types of per-node transformation descriptors:

A non-recursive descriptor for a given node N *stops* the generic algorithm from recursing into the subtree rooted in N. Instead, the algorithm expects the descriptor to transform that subtree by some other means. This type of descriptor is mainly useful if the shape of the subtree must change as part of the transformation (e.g. to cut off the subtree) or if the subtree should be transformed with a different transformation. The generic part of the algorithm in this case only accesses a given child and expects to be called by it in turn to transform possible child nodes.

A recursive descriptor causes the generic algorithm to *continue* its recursion down the tree. It expects the algorithm to take care of transforming all of its children and requires the transformed children as input to the node-local transformation.

The descriptors have to provide two pieces of functionality: At compile time, they must be able to compute the transformed type of a node. Depending on whether the descriptor is recursive, this job is handled either by a simple, fixed `typedef` (non-recursive descriptors) or a TMP that expects the types of the transformed children as input (recursive descriptors). Similarly, at run time they are used to construct an instance of the transformed node from the original node and – in case of the recursive descriptors – the already transformed child nodes.

Listing 5.4 shows the general structure of a nonrecursive transformation descriptor. It demonstrates one unfortunate peculiarity of the interface that is due to backwards compatibility: In existing PDELab code, the user typically creates objects like global and local function spaces as local variables. At the same time, TypeTree stores all children in dynamically allocated memory (as a `std::shared_ptr`). We thus need two different versions of the run time transformation code: One that accepts a stack object (as a const reference) and creates the transformed node on the stack as well, and a version that operates on heap-allocated objects in the form of `std::shared_ptr`.

In many cases, the per-node transformations really are local to the node, i.e. their outcome only depends on the node itself and not on its children. This usage scenario is handled by recursive transformation descriptors. These descriptors depend on the generic transformation algorithm to transform the children of the current node before invoking the local transformations. As the type of a node depends on its children, the algorithm determines the transformed node type by calling a TMP on the descriptor, passing it the transformed child types. Likewise, the run time `transform()` methods of the descriptor expect the algorithm to handle the creation of the child objects and pass the transformed nodes to the `transform()` function. Different node types (`CompositeNode` / `PowerNode`) use different conventions for passing around information about their children, and for that reason the API for the recursive descriptors is different for each node type. For detailed information, we refer to the library documentation; in the following, we only show an example for the `VariadicCompositeNode` in Listing 5.5.

5.5.2 Transformation Descriptor Registry

When performing a tree transformation, TypeTree relies on a transformation descriptor registry to look up the per-node transformations. This registry uses a function-based registration scheme, which will be described in detail in the next section. While this registry is never stored explicitly (it is a compile time concept, and descriptors are registered by declaring functions with a specific signature), there

Listing 5.4 — Transformation descriptor with user-controlled transformation of child nodes

```
template<typename SourceNode, typename Transformation>
struct NonRecursiveNodeDescriptor
{
  // indicates a non-recursive descriptor
  static const bool recursive = false;

  // here, the result type has to be calculated
  typedef ... transformed_type;

  // storage is usually in a shared_ptr (depends on node type)
  typedef shared_ptr<
    transformed_type
    > transformed_storage_type;

  // create instance of transformed type (stack -> stack)
  static transformed_type
  transform(
    const SourceNode& s
    const Transformation& t
    )
  {
    return transformed_type(s,t);
  }

  // create instance of transformed type (heap -> heap)
  static transformed_storage_type
  transform_storage(
    shared_ptr<const SourceNode> s,
    const Transformation& t
    )
  {
    return make_shared<transformed_type>(s,t);
  }

};
```

is nevertheless a canonical API that implements the double dispatch mechanism for looking up a transformation descriptor. Given a `Node` and a `TreeTransform`, the node transformation descriptor can be looked up by

```
typedef LookupNodeTransformation<
  Node,
  TreeTransform,
  typename Node::ImplementationTag
  >::type NodeTransformationDescriptor;
```

In order to register a transformation descriptor `descriptor_type` for a specific combination of a tree transformation `TreeTransform` and an implementation tag `ImplementationTag`, the user *declares* an overloaded version of the function

Listing 5.5 — Transformation descriptor with automatic transformation of child nodes

```
template<typename SourceNode, typename Transformation>
struct NonRecursiveDescriptor
{
  // indicates a recursive descriptor
  static const bool recursive = true;

  // return type is now a TMP that depends on child types
  template<typename... TC>
  struct result
  {
    typedef typename ... type;
    typedef shared_ptr<type> storage_type;
  };

  // for this node type, children are passed as variadic template argument list
  template<typename... TC>
  static typename result<TC...>::type
  transform(
    const SourceNode& s,
    const Transformation& t,
    shared_ptr<TC>... children
    )
  {
    return typename result<TC...>::type(s,t,children...);
  }

  template<typename... TC>
  static typename result<TC...>::storage_type
  transform_storage(
    shared_ptr<const SourceNode> s,
    const Transformation& t,
    shared_ptr<TC>... children
    )
  {
    return make_shared<typename result<TC...>::type>(s,t,children...);
  }

};
```

```
descriptor_type registerNodeTransformation(
  Node*,
  TreeTransform*,
  ImplementationTag*
  );
```

in a namespace where it can be found via argument-dependent lookup (ADL). Note that this function does only have to be declared, it does not need a definition. For example, in PDELab the `LocalFunctionSpace` is generated from a function

space via the transformation `gfs_to_lfs`. The transformation descriptor for this transformation and the `CompositeGridFunctionSpace` is registered by placing the following overload in the namespace `Dune::PDELab`:

```
template<typename CompositeGFS, typename Params>
CompositeLFSDescriptor<CompositeGFS,Params>      // result of the lookup
registerNodeTransformation(
  CompositeGFS*,
  gfs_to_lfs<Params>*,
  CompositeGridFunctionSpaceTag
  );
```

The specific transformation `gfs_to_lfs` is fairly complex and the logic inside the transformation descriptor requires additional parameters. In this example, we package those parameters into a template parameter of the transformation tag. That way, we can still require a specific outer type for the transformation so that our registration matches all transformations with this tag, irrespective of the nested parameters, but the registered descriptor is never considered for other types of transformations.

5.6 Tag Dispatch With Polymorphic Meta Functions

The utility of arranging tags into hierarchies to achieve compile time polymorphism has been recognized for a long time; the iterator tags defined by [C++/24.4.3] are often cited as a canonical example. In that clause, the standard also explains how these tags can be used to provide optimized algorithms by overloading function calls on the tag type:

```
// default algorithm
template<typename It>
void foo(It begin, It end, std::forward_iterator_tag);

// optimized algorithm for random access iterators
template<typename It>
void foo(It begin, It end, std::random_access_iterator_tag);

// user-visible function that performs tag dispatch
template<typename It>
void foo(It begin, It end) {
  foo(begin,end,typename iterator_traits<It>::iterator_category());
}
```

As the iterator tag will automatically be cast to its most-derived base class for which an overload of `foo` can be found, this technique easily accommodates new tags, which only need to be placed at the appropriate place in the tag hierachy, and will fall back to an implementation for the base class of a given tag if there is no function overload for the tag itself. This creates a dispatch mechanism that closely resembles dynamic polymorphism in that it selects the executed code based

on the argument types, but it happens at compile time and is thus called *static polymorphism* or *compile time* polymorphism.

In the following, we introduce a new type of meta functions that makes this type of polymorphism available to type calculations in addition to run time code.

As shown in Section 2.4.2, C++ meta functions are usually implemented as templated `structs`, where the arguments to the function are passed as template parameters and the return type can be retrieved as a nested **`typedef`** inside the **`struct`**. Control flow within the meta function then happens by (partially) specializing the **`struct`**.

In order to replicate the polymorphic properties of the iterator example at the beginning of this section, we instead *overload* a function signature, with the meta function parameters now passed as the types of the actual function arguments and the result of the meta function call encoded in the function's return type. This is possible because [C++/8.3.5.3] permits overloaded functions to have different return types, and while this feature may be confusing in normal C++ code, it fits perfectly for our application.

Let us consider a very simple example, which lets us register a descriptor based on a single tag type. If we assume that the meta function is called `dispatch`, we provide an overload that connects the input value `X` with the return type `Y`:

```
1 // no initial declaration of dispatch required for function overloads
2
3 // register X -> Y relation
4 Y dispatch(X*); // only declaration
```

Note that instead of directly specifying the tag in the argument list, we use a pointer; this allows `X` to be an incomplete type at the time the meta function is called. As explained earlier, this function only needs to be declared; a possible definition will never be used.

Invoking this meta function is slightly more involved than invoking a **`struct`**-based meta function. First of all, we need a way to capture the return type of a function in a **`typedef`**. For this purpose, we use the C++11 keyword **`decltype`**, which returns the result type of its argument (cf. Section 2.4.3). With its help, we can invoke the meta function through[1]

```
1 // utility struct to generate a value of type T*
2 template<typename T>
3 T* declptr();
4
5 // helper struct to encapsulate invocation syntax
6 template<typename Tag>
7 struct invoke_dispatch {
8   typedef decltype(dispatch(declptr<Tag>())) type;
```

[1] For older compiler versions that lack C++11 and **`decltype`**, it is possible to use the non-standard extension `__typeof__`. Apart from some semantic corner cases not relevant in this context, their behavior is identical.

```
9  };
10
11 typedef typename invoke_dispatch<U>::type result;
```

Note that in order to create the faked call to `dispatch()` inside **`decltype`**, we need to fabricate an argument of type `Tag*`. While we could do so by writing **`static_cast`**`<Tag*>(`**`nullptr`**`)`, the utility function `declptr()` conveys its meaning much clearer. In the example above, we have wrapped the real dispatch function in a traditional meta function called `invoke_dispatch` which provides an interface more familiar to users and encapsulates the slightly involved call to `dispatch`.

In this example, the overall result looks very similar to a "traditional" meta function, but the two approaches actually use different lookup mechanisms for resolving the general meta function to the correct version for its arguments: Traditional meta functions rely on template specialization, while our function-based meta functions are based on function overload resolution. These two mechanisms match their arguments according to different rule sets:

- An instantiated template is looked up by first searching for a complete specialization [C++/14.7.3], progressively widening the search scope to partial template specializations [C++/14.5.5.2] and finally falling back to the primary template definition.
- Function overload resolution as defined by [C++/13.3] is considerably more complex, in particular because [C++/13.3.2] permits implicit conversions of the function arguments to find a viable function. As there are multiple ways for the compiler to perform an implicit conversion, there can be a large number of candidate functions obtained by applying different conversions to the arguments. Consequently, the rule set in [C++/13.3.3] that prioritizes those candidate functions is considerably more complex than the class template instantiation rules, especially in the presence of function templates or even function template specializations.

Importantly, the template instantiation mechanism always matches very specific argument types (or, in the case of partial specializations, possibly specific templates with arbitrary inner template arguments). Due to the very nature of templates in C++, the logic responsible for finding a matching specialization does not take into account any relationships between types, such as inheritance or conversion operators This restriction requires the explicit registration of every single supported tag by specializing the dispatch meta function. Function overload resolution on the other hand knows about type relations and will in particular cast a type to one of its base classes if there is a function overload that takes the base class as a parameter, but no overload for the type itself.

In the context of TypeTree, this property of the function overload resolution process makes it possible to create a hierarchy of `ImplementationTag`s similar to the iterator example at the beginning of this section. While the base tag will

have to be registered for all transformations, derived tags only need to provide more specialized transformation descriptors if their behavior should differ from the base tag for this specific transformation. This functionality is used by PDELab to implement a `VectorGridFunctionSpace`, which is a special case of the `PowerGridFunctionSpace` for variables that represent spatial vectors (e.g. a velocity). The implementation tag of this `VectorGridFunctionSpace` inherits from the tag of the `PowerGridFunctionSpace` and is thus able to reuse the majority of its implementation. It only overloads a very small number of transformations to e.g. output its values as VTK vector data, while the normal `PowerGridFunctionSpace` outputs its children as separate scalar data sets.

Another advantage of function overloading is the additional flexibility in where to place the registration declarations:

- Class template specializations have to be placed in the same scope as the primary template [C++/14.5.5], which is inconvenient in the context of meta functions because the namespace of the specialized meta function will often be different from the application namespace with the user types.
- On the other hand, function overloads will be found in a much larger number of scopes as long as the function call is not qualified with a specific scope. In addition to the scope of the call site, ADL will also consider the scopes of all arguments (both regular and template ones) when looking for candidate functions, which enables users to place the function declaration registering the dispatch pattern directly into the application namespace(s).

5.7 Applications

TypeTree was initially written as a utility library for PDELab, where it is of central importance for a multitude of tree-based data structures, chiefly the function spaces and several derived objects like the tree of `Ordering` objects. In the following, we highlight several examples that employ some of the more advanced features of the library.

5.7.1 Proxy Nodes

Within PDELab there are several scenarios that involve taking an existing tree node and extending and / or modifying its behavior while retaining the structure of the underlying tree. In core PDELab, subspaces are the most visible application of this pattern: they make it possible to extract a subtree of a larger space and make that subtree aware of the overall tree structure. This "ancestry" knowledge is normally not available in TypeTree trees, but is required to map subspace-local DOFs to the global space. At the same time, the remainder of the function space API of the subspace including its subtree structure can be reused from the original tree node. In order to simplify the implementation of this kind of *proxy objects*,

TypeTree provides a `ProxyNode` that stores an existing node and mimics its tree node behavior (it will have the same `NodeTag` and offer the same API, which simply forwards to the proxied node). This encapsulation normally avoids the need to reimplement the user-specific portion of the proxy for each different type of tree node. Instead, most of the time it will be sufficient to have a single implementation and inherit from `ProxyNode` as in this trivial example:

```
template<typename Node>
struct Proxy
  : public ProxyNode<Node>
{
  Proxy(shared_ptr<Node> node)
    : ProxyNode<Node>(node) // forward original node to base class
                            // that handles storage etc.
  {
    this->proxiedNode();    // derived classes have access to the proxied node
  }

  ...
};
```

This proxy will work for both leaf nodes as well as composite nodes or any other type of TypeTree node. Internally, `ProxyNode` is an interface class that provides the basic node interface shared by all node types. Additional functionality specific to the function of the node in the tree structure is then injected by inheriting from a mixin class that depends on both the proxied type and its node tag. The library ships with appropriate specializations of this mixin class for the default node tags, which contain the additional API for static and dynamic child access.

The mixin-based design makes it very easy to add support for additional node tags. For example, if a user has created a new type of node with node tag `FooNodeTag`, it is only necessary to provide the partial specialization

```
template<typename Node>
struct ProxyNodeBase<Node,FooNodeTag>
{ ... };
```

Afterwards, our general `Proxy` class from above will automatically support the TypeTree nodes of type `FooNode`.

5.7.2 Filtered Nodes

The subproblems and couplings introduced in Chapter 4 define a subspace for the associated residual form. This subspace is given as a tuple of child indices. If the tuple only contains a single child index, the subspace can be implemented as a `ProxyNode` on top of that child, but if the space combines multiple children, we need to synthesize a new tree node that presents the selected children in the correct order. For this purpose, TypeTree introduces another powerful utility called `FilteredCompositeNode` that applies a compile time filter to the children of an interior node. The library also ships with a number of predefined filters;

in order to select and reorder children based on a subspace index tuple, we can use the `IndexFilter`. For example, if we need a tree node that contains children $(4, 1, 3, 2, 3)$ of an existing node, the filtered proxy node can be constructed as follows:

```
template<typename Node>
struct Filtered
  : public FilteredCompositeNode<
             Node,                    // underlying node
             IndexFilter<4,1,3,2,3>   // select children of node in this order
             >
{
  typedef FilteredCompositeNode<
    Node,
    IndexFilter<4,1,3,2,3>
    > Base;

  Filtered(shared_ptr<Node> node)
    : Base(node)
  {
    this->unfiltered();              // access unfiltered node
  }

  ...
};
```

As this example shows, it is even possible to duplicate a child in the filtered node, but keep in mind that the filter does not really copy the node; in the example above, the 3rd and the 5th child of the filtered node point to the same object in memory.

The filter passed to the `FilteredCompositeNode` is actually a TMP that is invoked when the filtered node is instantiated; during its invocation, it has access to the types of the underlying, unfiltered node and its children. It returns a tuple of index pairs $(i_{\text{new}}, i_{\text{old}})$ encapsulated in a struct that conforms to the concept of a `FilterResult`; a precise definition of that concept with type names etc. can be found in the library documentation. For the example above, the result of calling the `IndexFilter` will be $((0, 4), (1, 1), (2, 3), (3, 2), (4, 3))$.

By implementing the filter as a TMP instead of a simple list of indices, it becomes possible to perform filtering on almost arbitrary criteria (e.g. remove all children that are leaf nodes, sort children by some user-defined property, ...). On the other hand, the `FilteredCompositeNode` is oblivious to the exact semantics of the filter; there is only a single implementation that is shared across all types of filters and interior nodes. Note, however, that filtering a `PowerNode` removes the extended interface of that node type, as the `FilteredCompositeNode` always mimics a `VariadicCompositeNode`, as adding special support for a dynamic child interface was not needed for our applications and seemed too much of a special case to warrant the additional implementation complexity.

CHAPTER 6

Flexible Control of Vector and Matrix Layout

Throughout the preceding chapters we have focused on the intricacies of formulating multi physics problems as well as assembling their residuals and Jacobians, but until now we have ignored the actual layout of the vectors and containers representing those objects. That layout is governed by the way we enumerate the basis functions, and that order does in turn influence the behavior of the numerical solver we use for solving the (linear or non-linear) equation system formed by the residual and the Jacobian.

The most important problem that these equation systems pose is their immense size – outside of the sheltered haven of mathematical method development, millions of equations are considered a small problem in this context, and large HPC simulations currently operate on problems on the order of 10^{12} DOFs. These enormous problem sizes make it impossible to naively store the problem matrices in a dense representation – even for a small problem with 10^6 DOFs, a dense double precision matrix would require ≈ 7.3 TiB of storage, which would in turn render the solution of such a problem on current workstation hardware completely unfeasible. On the other hand, typical FEM matrices are very sparse, so they are usually stored in formats that exploit this sparsity, e.g. block compressed row storage (BCRS).

For realistic problem sizes, the linear equation systems encountered (either directly or after linearization of a non-linear system) are solved using iterative solvers. Iterative solvers often exhibit performance characteristics that strongly depend on the structure of the matrices / vectors, a structure that is created by the problem assembly algorithms. Specifically, this influence is present through three distinct, but interdependent mechanisms:

- Modification of the *intra-space order* of DOFs allows for controlling features like the bandwidth of the assembled matrix, an important factor for the performance of some iterative solvers / preconditioners.
- When *merging product spaces* on a common tessellation, there are multiple approaches, which allow for different interpretations of and solution techniques for the overall problem.
- Somewhat related to the previous point, specific DOF layouts for certain problems benefit from the use of *block matrices / vectors* to expose the problem structure to the algebraic solver and improve its performance.

In this chapter, we present an iterative mechanism for generating enumerations of the basis functions of function spaces based on the tree structure of the underlying function spaces and discuss the implementation of this mechanism within PDELab. As index mapping is a core functionality of a PDE assembly framework, our implementation in this case does not work as an optional add-on, but completely replaces the earlier, more limited mapping functionality throughout the framework. Consequentally, the discussion in this chapter is not limited to multi domain problems, but also applies to simpler problems like the Navier-Stokes equation presented in Section 2.1.4.

6.1 Ordering Degrees of Freedom in Finite Element Bases

A discrete FE function space V is defined in terms of its underlying mesh: It is spanned by a finite number of basis functions $\{\varphi_i\}_{i=0,\dots,n-1}$, each associated with a specific mesh entity (note that there may be multiple basis functions associated with a single entity, e.g. for P_k spaces with $k > 2$), and any solution $\psi \in V$ in that space can be represented by a *DOF vector* $\boldsymbol{u} \in \mathbb{R}^n$ that implies a linear combination of the basis functions:

$$\psi = \sum_{i=0}^{n-1} u_i \varphi_i. \tag{6.1}$$

In order to utilize this equation for a concrete function space, we have to somehow uniquely enumerate the basis functions. While this problem may seem trivial for simple cases like a scalar Q_1 space (there is one basis function per grid vertex; just iterate over the vertices in your grid data structure and number the functions in iteration order), we will see that there are a number of choices once we encounter product spaces or more complicated finite elements. In general, it is not possible to pick a single enumeration scheme that is optimal in all situations; instead, we need to be able to adapt the enumeration to the problem, the discretization and the linear solver.

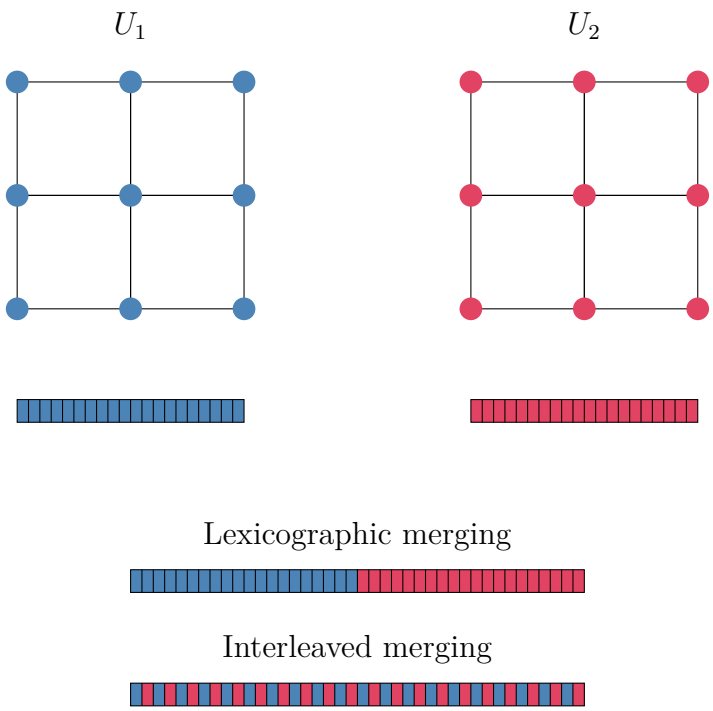


Figure 6.1 — Lexicographic and interleaved DOF merging for two Q_1 spaces. Top: Meshes showing mapping of DOFs to grid entities and resulting DOF vectors. Bottom: Results of different merging strategies.

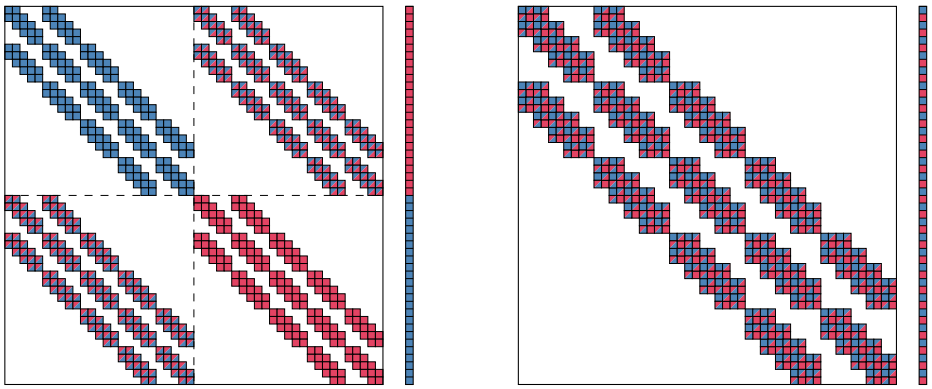
6.1.1 Merging Index Ranges in Product Spaces

When assembling a problem with multiple variables, there are two fundamental approaches to merging the already ordered basis functions of the individual components into a global basis:

- We can simply concatenate the complete bases of the children, which we call a *lexicographic order*. This creates a single contiguous range of basis functions for each subspace.
- If the child spaces have a similar structure (e.g. both children are Q_1, or both children are DG spaces), we can *interleave* the child bases according to a simple ratio rule.

Figure 6.1 illustrates these two approaches by means of a product space of two Q_1 spaces in 2D. In this case, the interleaved approach works well, as it groups the basis functions of the merged space by their associated grid entities.

System matrices for problems with lexicographically merged product spaces contain a single large diagonal block for each subspace, along with off-diagonal blocks that



(a) Lexicographic merging (b) Interleaved merging

Figure 6.2 — Matrix and vector layouts for different subspace merging methods of a 2D product space of two Q_1 spaces

describe the coupling behavior between pairs of subspaces:

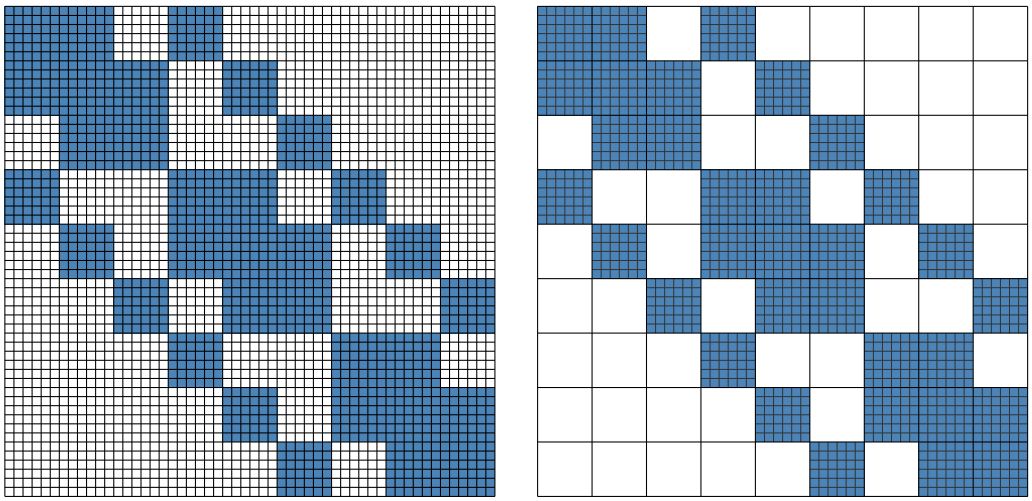
$$\begin{pmatrix} A_1 & C_{12} \\ C_{21} & A_2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}. \tag{6.2}$$

On the other hand, matrices that arise from interleaved merging mimic the structure of the smaller matrices corresponding to each subproblem (A_1 and A_2 above). Instead of a scalar value, each matrix entry becomes a small, dense matrix (of size 2×2 in the example above). Figure 6.2 illustrates this principle for a $Q_1 \times Q_1$ space on a 2D mesh with 4×4 cells; note how the interleaved matrix on the right looks like a scaled-up version of the individual blocks on the left. It also demonstrates the association between the block structures of the matrix, the solution vector and the right hand side; the structure of the solution is identical to the column block structure of the matrix, while the right hand side mirrors its row block structure.

6.1.2 Block Structured Vectors and Matrices

While the data structures of the computer implementations for vectors and matrices do not necessarily have to take this block structure into account, there are advantages to using block-aware containers:

- Many advanced iterative solvers are based on a decomposition of the system matrix into macro blocks as shown in (6.2) and do not operate on the global matrix, but on those blocks (e.g. block preconditioners and domain decomposition methods like Dirichlet-Neumann). Storing those blocks as directly accessible objects inside the global containers avoids expensive copy operations when accessing those blocks. This is of particular importance in the matrix case because (a) the matrices contain much more data than



(a) Matrix pattern without blocking (b) Matrix pattern with blocking

Figure 6.3 — Sparse matrix pattern of a 2D DG discretization, blue color denotes non-zero matrix entries. In the non-blocked case, the pattern has to store a column index for each matrix entry; in the blocked case, it is sufficient to store one index per 6×6 matrix bock, reducing the pattern storage overhead by a factor of 36.

the vectors, and (b) FEM matrices are extremely sparse and thus stored in compressed form; extracting a block out of the global, compressed matrix is a very expensive operation in terms of memory and processing time.

- Compressing the matrices requires storing the location of populated matrix entries. A sparse block matrix only needs to store the location of the blocks; this technique can greatly reduce the size of the sparsity pattern. Figure 6.3 shows the effect of this optimization using a DG discretization, where the 6 DOFs per grid cell are placed into a single matrix block of size 6×6; consequently, the pattern size was shrunk by a factor of 36. If the matrix uses double precision and 64 bit indices, these savings cut the size of the overall matrix almost in half.
- Block-aware preconditioners like block Jacobi or block Gauss-Seidel are extensions of their scalar counterparts to block-structured matrices. If this block structure is not reflected in memory, implementing those types of preconditioners becomes much more difficult, and performance suffers.

The examples shown here are based on a single level of blocking, but it is of course possible to have a nested block structure, especially when considering problems with highly structured function spaces.

As explained above, block structures naturally arise when merging index ranges: Every time two (or more) index ranges are combined, we can decide whether we also want to explicitly represent this block structure in our data structures. Thus

the two processes are intrinsically connected to each other and linked to the interior function space nodes, where index ranges have to be merged. For these reasons, PDELab attaches this merging and blocking information to the interior nodes of the function space tree, which happens by supplying two template parameters that specify an `OrderingTag` and a `VectorBackend`, a mechanism that will be explained in detail in Section 6.5.

6.2 Structure-preserving DOF Indexing

In the previous section we have established that there are often multiple ways of enumerating the basis functions of a function space and that the choice of enumeration and block structure depends on the problem at hand. For that reason, it should be possible to pick different ordering strategies for a given function space. In order to do so, we separate the concept of a function space as a container of basis functions from that of the *ordering*, a new component that controls the layout (order and block structure) of the DOF vectors for the space (cf. (6.1)).

This layout component should be able to take the unordered set of basis functions of a function space and generate an enumeration by applying a set of merging and blocking operations. Basically, it is a map from the set of basis functions to the integer range $0, \dots, N$, where N is the number of basis functions. In the future, we will denote such an ordering by $\mathcal{M}$. In order to implement this map, we need a way to uniquely identify each basis function. To do so, recall that finite element bases associate each basis function φ with a specific grid entity τ; the basis functions of a scalar function space can thus be identified by a pair (k, τ), where k is an integer used to enumerate multiple basis functions attached to the same grid entity τ.

When dealing with a composite space (represented as a tree), we need to further disambiguate the basis functions of the individual leaf spaces, which we do by extending the identifier (τ, k) that is unique for a leaf space by appending the tree path from that leaf space to the root of the function space tree:

Definition 6.1 (`DOFIndex`)**.** *Let V a composite function space with associated structural tree $\mathcal{B}$, underlying mesh $\mathcal{T}$ and basis $\Phi = \{\varphi_i\}_{i \in 0,\dots,N-1}$. Furthermore, let V_j a leaf subspace of V with tree path $j = (j_1, \dots, j_n)$ in $\mathcal{B}$. Finally, let $\varphi \in \Phi$ the k-th basis function of V_j that is associated with mesh entity τ. Then the* `DOFIndex` *$\mathcal{D}(\varphi)$ of φ is defined as the tuple*

$$\mathcal{D}(\varphi) = \Big(k, \mathcal{G}(\tau), j_n, \dots, j_1)\Big),$$

where $\mathcal{G}$ denotes an appropriate encoding of the identity of τ to a tuple of integers such that $\mathcal{G}(\tau) \in \mathbb{N}^{n_g}$ with some fixed n_g.

Remark 6.1. In our implementation within PDELab, $\mathcal{G}$ maps a mesh entity to a pair of indices, one for the geometry type of the entity and one for its index within the current index set (note that DUNE provides separate index ranges for

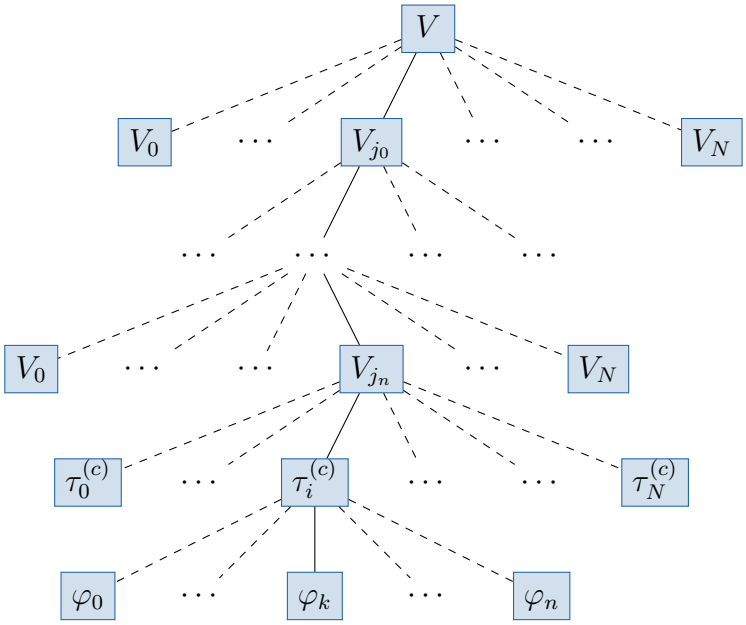


Figure 6.4 — Tree of basis functions for a hierarchically structured multi-component space V. The pictured basis function φ_k can be addressed by the index tuple $j = (k, (c, i), j_n, \ldots, j_0)$.

distinct geometry types, cf. Section 2.3.2 and [18, 19] for further information on entity enumeration in DUNE). However, for the purpose of the following discussion, we will assume that $\mathcal{G}$ maps all mesh entities into a single, contiguous index range $(0, \ldots, |\mathcal{T}| - 1)$. Instead of $\mathcal{G}(\tau)$, we write τ_k to indicate the grid entity with index k.

Note that the `DOFIndex` for a given basis function ϕ on grid entity T can be trivially constructed without any knowledge about the size of the global function space. At the same time, it encodes a large amount of information about the position of the basis function with regard to the function space structure, making it an ideal input value for the index transformation $\mathcal{M}$. The `DOFIndex` arranges the basis functions of a finite element space into a tree that mirrors the hierarchical structure of the space, as shown in Figure 6.4: The leaves of the function space are extended by one node per grid entity; the individual basis functions form the new leaves of the tree and become children of their associated grid entity.

Given a fixed traversal order for the mesh entities, this tree directly induces one possible enumeration of the basis functions. It can be obtained by performing a depth-first traversal of the tree and numbering the basis functions in ascending order as they are encountered. Note that we are in general not interested in storing this global enumeration; typical FEM assembly operations only involve a small number of basis functions at any given time. Thus, we need an efficient method

to calculate the enumeration value for the `DOFIndex` $\mathcal{D}(\varphi)$ of an arbitrary basis function φ on the fly based on its `DOFIndex` representation.

6.3 Using Multi Indices for Vector and Matrix Access

Up until now, we have assumed that the output of the ordering map $\mathcal{M}$ will yield a flat consecutive ordering, i.e. that it maps to a consecutive, zero-based range $(0, \ldots, N)$. If the simulation data is stored in block-structured vectors and matrices, such a flat indexing scheme imposes a significant overhead because we have to somehow reconstruct the block-wise indexing before we can access an entry in such a containers. This involves calculating a block number i_b and an intra-block index i_s. A more efficient addressing scheme in this case will use a map $\mathcal{M}$ that directly creates a two-component index which encapsulates those two integers in a tuple (i_b, i_s). If there are multiple levels of blocking, the length of this tuple increases accordingly. Figure 6.5 illustrates this idea for a simple vector with one level of blocking and a fixed block size. In this case, it is still possible to recover the block number and the intra-block index using simple mathematical operations. If the blocks are of variable size, however, this is no longer possible, and different means for identifying the correct block have to be found, e.g. a binary search in a lookup table with block offsets.

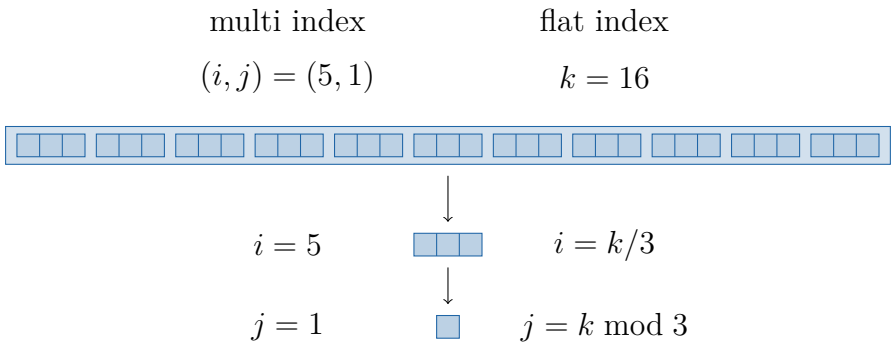


Figure 6.5 — Access to a simple block structured vector with block size 3 using flat indices and multi indices

In Section 6.1.1, we have shown that the block structure of the vectors and matrices results from operations on the nodes of the function space tree. As the `DOFIndex` essentially encodes this tree structure, it contains all necessary information to design very efficient algorithms that directly convert the `DofIndex` to a multi index like the one in the example above. This is the approach taken by the DOF ordering framework in PDELab: The output of the ordering map $\mathcal{M}$ will be a multi index tailored to the selected block structure of the data structures.

6.4 Nodal Operations for DOF Numbering Generation

As we have seen previously, the (multi) index for a basis function φ generated by an ordering $\mathcal{M}$ is constructed in an iterative fashion by starting at the leaf space associated with φ and then walking up the function space tree, performing the necessary merging (and, if requested, blocking) operations at each node.

It is thus possible to break the overall ordering map $\mathcal{M}$ down into a series of nodal operations M_i associated with individual entries of the `DOFIndex`. In the following, we assume that such an operation may only depend on the current tail entry of the output tuple (i.e. the container index) and the child index of the current node in the `DOFIndex` tree. For example, if the incoming container index is (i_1, i_2), i.e. a tuple of length 2, and the `DOFIndex` entry at the current position is j, a nodal operation M can take on one of two forms:

$$M^f\Big((i_1, i_2), j\Big) = \Big(i_1, m_f(i_2, j)\Big) \qquad \text{(flat transformation)}$$
$$M^b\Big((i_1, i_2), j\Big) = \Big(i_1, i_2, m_b(i_2, j)\Big) \qquad \text{(block transformation)}$$

As we are only interested in the last entry of the container index, we will omit the preceding entries and just write $(\dots, i)$ in the future.

At the leaf nodes of the `DOFIndex` tree, we only consider the following trivial transformation: Given an initial, empty container index tuple $()$ and a `DOFIndex` $(k, \mathcal{G}(\tau_l^{(c)}), j_n, \dots, j_1)$, the transformation M_0 is given by $M_0((), k) = (k)$. The remaining node types are more complicated; the corresponding transformations are introduced in the following sections.

6.4.1 Grid Entity Nodes

The merge operation at a grid entity node can be understood as a special case of the interleaving approach. In general, we always want all basis functions associated with a single grid entity to be grouped together, as those basis functions will usually couple with each other. Thus we define the grid entity transformation by

$$M_{\mathcal{T}}^f\Big((\dots, i), \tau_l\Big) = \left(\dots, i + \sum_{j=0}^{l-1} s_j\right), \tag{6.3}$$

where s_j denotes the number of basis functions associated with the mesh entity τ_j. Equivalently, the blocked version M_0^b creates a multi index of length 2 and is given by

$$M_{\mathcal{T}}^b\Big((\dots, i), \tau_l\Big) = \Big(\dots, i, l\Big). \tag{6.4}$$

6.4.2 Lexicographic Merging

Lexicographic merging groups the basis functions by their associated child space and then sorts these blocks according to the order of the subspaces in the current tree node. Given a composite space V with direct children $(V_j)_{j=0,N-1}$, the transformation M_L^f that maps the index $(\dots, i)$ from child space V_j into the index range of V is given by

$$M_L^f\Big((\dots, i), j\Big) = \left(\dots, \left(i + \sum_{k=0}^{j-1} |V_k|\right)\right), \tag{6.5}$$

where $|V|$ denotes the dimension (and thus the size of the basis) of V. This strategy does not make any assumptions about the internal structure of or the relationships between subspaces and can thus be applied to any type of multi-component space. This combination of generality and ease of implementation makes lexicographic merging the standard algorithm implemented by almost all modern FEM frameworks [32, 15, 81].

The blocked version M_L^b of this transformation is again very simple: It just appends the child index j to the existing multi index, which yields

$$M_L^b\Big((\dots, i), j\Big) = (\dots, i, j). \tag{6.6}$$

6.4.3 Proportional Interleaving

Problems like the multi-component transport example from Section 4.2.3 can greatly benefit from grouping the values of the per-component concentrations in small per-entity vectors, which we can then directly invert using a block-aware preconditioner. Figure 6.2(b) shows an example of a matrix created by the proportional interleaving of two spaces of identical size.

In order to create this mapping for a composite space V with direct children $(V_j)_{j=0,N-1}$, each child space V_j has to be associated with an integer multiplicity $f_j \in \mathbb{N}$ such that $|V_j|/|V_k| = f_j/f_k,\ j,k = 0,\dots,N-1$. For example, consider a space V_1 that attaches 2 functions to each vertex and 4 functions to each edge. At the same time V_2 attaches 1 function to each vertex and 2 to each edge, yielding a fixed ratio of $2:1$. Assuming fixed multiplicities f_j, the proportional merging map M_P is given by

$$M_P^f\Big((\dots, i), j\Big) = \left(\dots, \left(i \bmod f_j + \sum_{k=0}^{j-1} f_k + \left\lfloor \frac{i}{f_j} \right\rfloor \sum_{k=0}^{N-1} f_k\right)\right), \tag{6.7}$$

while the blocked version is defined by

$$M_P^b\Big((\dots, i), j)\Big) = \left(\dots, \left(i \bmod f_j + \sum_{k=0}^{j-1} f_k\right), \left\lfloor \frac{i}{f_j} \right\rfloor\right). \tag{6.8}$$

Note that unlike the earlier transformations, the blocked version M_P^b in this case not only appends a new entry to the output index, but also modifies the last entry of the existing index.

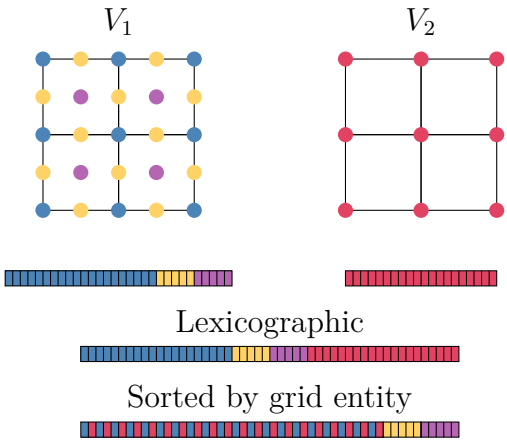


Figure 6.6 — Lexicographic merging and entity-based sorting for two structurally different function spaces. Note that the compatible parts of the spaces (associated with the grid vertices) have been merged, while the edge-related entries of the second space are simply appended in lexicographic fashion.

6.4.4 Grouping Composite Spaces by Grid Entity

Without a fixed integer ratio for the child space sizes, the static merging procedure described in the previous section will not work any more. The main reason for interleaving the values from the child spaces in this manner is to cluster the basis functions by grid entity in order to e.g. create a common diagonal matrix block that can then be used in a preconditioner like block Jacobi. In certain settings, this entity-wise clustering can be beneficial even without the fixed size ratio, e.g. for p-adaptive DG product spaces, where all degrees of freedom are associated with the cells; the p-adaptivity simply causes a different number of basis functions to be associtated with each cell. With the help of entity-wise clustering in the product space, we can retain the typical block structure of those spaces, albeit with blocks of varying size. Figure 6.6 illustrates this principle by means of a less useful example, an incompatible combination of a Q_1 and a Q_2 space, but demonstrates the general applicability of the principle

Looking back at the basis function tree in Figure 6.4, we can see that in it, all basis functions in a *leaf space* are grouped by their grid entities. If we want to perform this grouping at a higher level in the tree, we cannot do it by introducing a new type of nodal operation; instead, we have to reorganize the tree and move the grid entity node further up towards the root of the tree. The result of such a tree modification is shown in Figure 6.7, which depicts a tree that was derived from the one in Figure 6.4 by moving the entity nodes up to the third tree layer. In our implementation, this complex modification of the input tree is handled by a special TypeTree transformation.

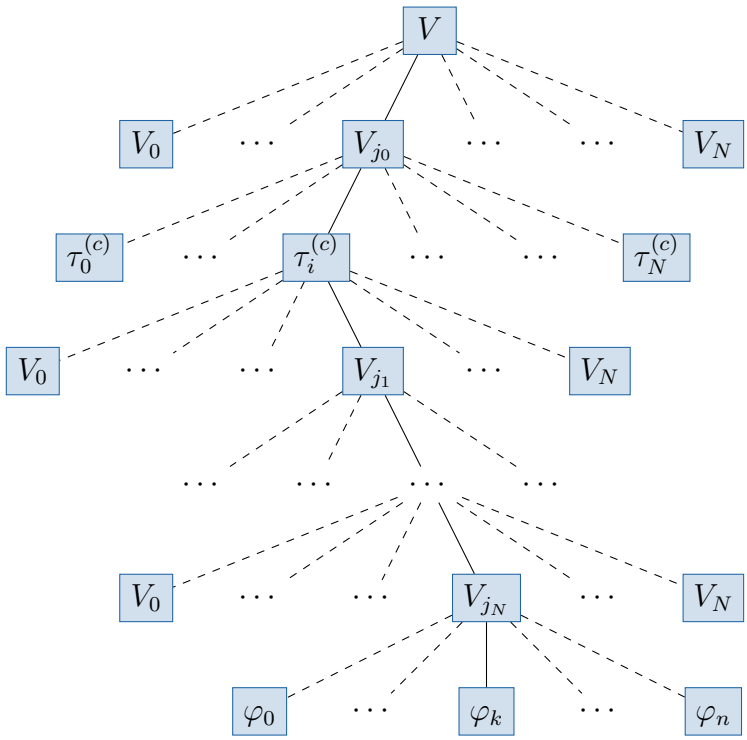


Figure 6.7 — Modified DOFIndex tree with grid entity information moved to the second index entry

6.5 DOF Ordering Library

As part of our work towards facilitating the implementation of block solver schemes in PDELab, in particular for the simulation of multi physics problems, we have created a new DOF ordering library for PDELab based on the concepts introduced in the previous section. In the following, we present the interface of that library and highlight relevant parts of its implementation.

The code of this implementation is contained in PDELab and is available under the same open source license as the remainder of that library (cf. Section 2.3).

6.5.1 Mesh Entity Processing

Performing the per-node transformations as described in Section 6.4 requires moving the grid entity information $\mathcal{G}(\tau_i)$ to a different position in the `DOFIndex` tuple. In order to simplify this process, the actual storage layout of the `DOFIndex` does not store $\mathcal{G}(\tau_i)$ as part of the index tuple; it instead contains a separate data structure for the grid index information.

A typical FEM mesh will contain a huge number of grid entities; creating a separate tree node for each entity is thus not viable. Our implementation instead creates a single object that extracts the grid entity information from the additional data

structure and uses it to perform the merging operation for all grid entities. Note that this creates a semantic mismatch between our theoretical concept and the actual implementation for all children of an entity node: Instead of a separate subtree for each grid entity, there is now only one single subtree that needs to handle the subtrees for all grid entities. Our implementation resolves this problem by introducing a second type of tree nodes called *local orderings*. These differ from standard orderings in that they are aware of the grid entity and store separate merging information for each entity.

6.5.2 Ordering Tree Creation

As we have previously seen, the tree with the nodal transformations mapping a `DOFIndex` to a `ContainerIndex` is closely related to the tree of function spaces created by the user. We can thus use a TypeTree *transformation* (cf. Section 5.5) to construct the tree of `Ordering` objects from the user-supplied function space tree. In order to determine the index mapping strategy at each node, the user has to annotate each function space node with additional tags. There are two choices to be made:

Merging For interior nodes, a merging strategy has to be chosen. Currently, the library supports lexicographic, proportional and per-entity merging. In the following examples, those will be symbolized by L, P and E, respectively.

Blocking The user has to decide whether or not the `ContainerIndex` should be blocked at the current level, i.e. whether a new block level should be appended to its end, which we denote by B for blocking and F for flat.

The tree of ordering objects is then constructed by means of a TypeTree transformations that traverses the function space tree and transforms each node based on those annotations. This transformation does not descend beyond grid processing nodes; the subtree associated with these nodes is created by means of a nested transformation that creates a tree of local orderings (see above). These orderings have a slightly different programming interface and always receive grid entity information in addition to the `DOFIndex`.

Choosing a Merging Strategy

The merging strategy is chosen by passing an `OrderingTag` to the template parameter list of the function space. For leaf spaces, the user can typically omit this tag because the default value of `DefaultLeafOrderingTag` will almost always be correct.

In contrast, composite spaces require the user to explicitly pass a tag describing the merging strategy. In the case of lexicographic merging and entity-blocked ordering, it is sufficient to specify this tag in the template parameter list:

```
typedef CompositeGridFunctionSpace<
  VectorBackend,             // controls blocking, see next section
  LexicographicOrderingTag,  // or EntityBlockedOrderingTag
  Children...
  > CompositeGFS;
```

In contrast, the proportionally interleaved ordering additionally needs to be told about the multiplicities f_i for each child space (cf. Section 6.4.3), which are stored as run time information and thus require an instance of the tag class. Usually, this instance can be created in-place in the constructor call of the function space:

```
typedef CompositeGridFunctionSpace<
  VectorBackend,
  InterleavedOrderingTag,    // choose interleaving
  Children...
  > CompositeGFS;
CompositeGFS composite_gfs(
  {2,1,4},                   // per-child multiplicities
  children...
  );
```

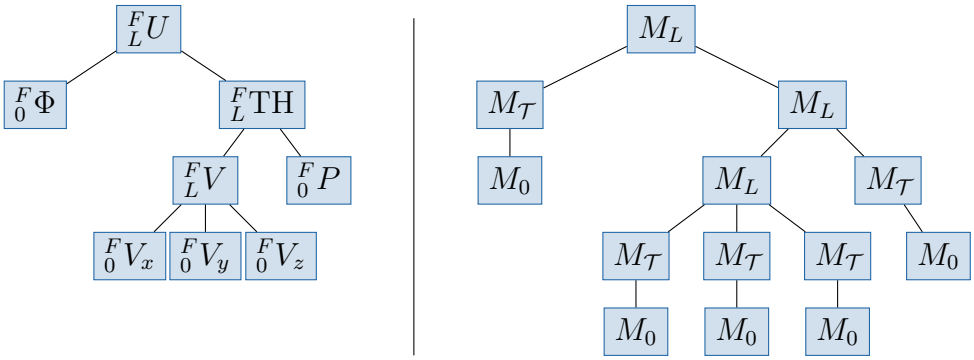
Here, we have used a C++11 `std::initializer_list` to provide the list of multiplicities, which avoids spelling out the tag type a second time in the constructor call.

In order to demonstrate the effects of those annotations, consider the function space tree for the Stokes–Darcy example from Section 4.2.2. Figure 6.8 depicts two versions of that tree on the left hand side, each with different merging and blocking annotations. On the opposite side you can see the respective `Ordering` trees induced by those annotations. Moreover, the example also shows the additional nodes $M_{\mathcal{T}}$ that have been inserted into the `Ordering` tree to handle the grid entity information.

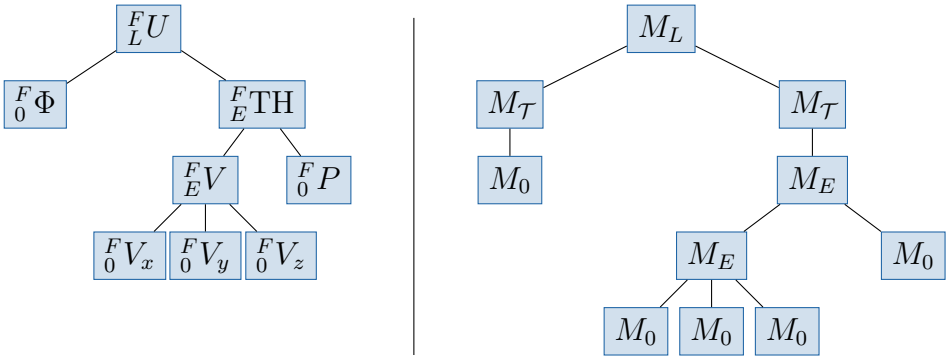
6.5.3 Automated Construction of Linear Algebra Containers from Annotated Function Spaces

The original backend architecture of PDELab always used a single level of DOF blocking, backed by the BCRS matrix implementation in the ISTL module. The resulting setup was a fixed vector and matrix structure with the globally constant block size as the only parameter.

Due to the lack of integration between the DOF mapping components (then part of the `GridFunctionSpace` tree) and the linear algebra backends, the old implementation required the user to specify the block size in multiple places. An inconsistent specification of the block sizes across these locations would result in compilation failures or wrong memory accesses at run time, causing programs to crash or – worse – deliver wrong results because values were not written to the correct vector and matrix entries.



(a) Lexicographic index merging for all spaces



(b) Entity-based merging for Taylor-Hood subspace, lexicographic merging at top level

Figure 6.8 — Ordering trees for two differently annotated Stokes-Darcy function space trees. The function space nodes on the left are annotated with their blocking behavior (here always F for *flat*) and their merging strategy (L = *lexicograpic*, E = *entity-based*). The resulting trees of `Ordering` objects are shown on the right-hand side. Note the upward movement of the mesh handling component $M_\mathcal{T}$ in the entity-blocked case.

On the other hand, we can infer all information about the nested vector and matrix structure and the associated block sizes from lower-level building blocks: The block structure of the ansatz and test space vectors can be constructed from the per-node blocking annotations of the underlying function spaces, and the block structure of the matrix can be obtained by taking the tensor product of those two vectors: for example, in a program that uses vectors with block size 6 for the ansatz space and block size 3 for the test space, the corresponding matrix has to consist of blocks of size 3×6.

As part of our changes to the DOF mapping infrastructure of PDELab, we have created a set of TMPs that implement this functionality at compile time; with this functionality, the user now only needs to specify the desired block structure when

creating the function space; the grid function space can then be used to automatically create and initialize correctly structured vectors; a similar functionality for matrices is provided by the `GridOperator`, greatly reducing the possibility for inconsistencies in their simulation setup.

The exact structure of the vectors and matrices is determined by the backend tags introduced in the next section.

Choosing a Vector Backend and a Block Structure

The function space template parameter for the `VectorBackend` serves a dual purpose, selecting both the backend library to use for the DOF vectors and their block structure. Currently, PDELAB only ships with a single vector backend tag called `ISTLVectorBackend`. In its default form, `ISTLVectorBackend<>` (all template parameters defaulted), blocking is disabled at the current level. If the user wants to create blocks, it is necessary to differentiate between two different scenarios:

- When creating a large number of small blocks (e.g. for interleaved merging), these are typically stored in a block vector with statically allocated blocks of fixed size (a `BlockVector<FieldVector<double,block_size> >`). Here, the block size has to be known at compile compile time and is passed as an additional template parameter:

```
typedef ISTLVectorBackend<
  ISTLParameters::static_blocking,
  block_size
  > SmallFixedSizeBlocksBackend;
```

  Note that because the block size is given as a template parameter, the associated vectors do not support blocks of variable size and are unsuitable for e.g. *p*-adaptive DG discretizations. This is a limitation of the underlying vector and matrix implementations in acISTL , which have no efficient support for variable size dense blocks.

- The large macro blocks associated with lexicographic merging are stored as a nested block vector of type

```
BlockVector<
  BlockVector<
    FieldVector<
      double,
      block_size
      >
    >
  >
```

  The blocks in this vector are dynamically allocated and may thus vary in size. Vectors of this type are created by specifying `dynamic_blocking` in the backend tag:

```
ISTLVectorBackend<
  ISTLParameters::dynamic_blocking
  >;
```

When users create a DOF vector for a function space, a TMP traverses the underlying tree of spaces, inspects the vector backend tags and automatically constructs the vector type. The algorithm also checks for inconsistencies across the tree branches (e.g. different static block sizes) and raises a compile time error if it is unable to deduce the vector type. The native ISTL vector is then wrapped in a PDELab container that conforms to the PDELab backend interface. This wrapper is responsible for mediating between the PDELab interface (e.g. entry access with a multi index `ContainerIndex`) and the ISTL API.

The structure of the system matrix is deduced in similar fashion by combining the block structures of the ansatz and test space vectors, which determine the column and row blocks, respectively. Nevertheless, the user has to specify a matrix backend descriptor when creating a `GridOperator`. This descriptor does not influence the matrix type (as it is deduced from the solution and residual vectors), but is currently used to switch between two different methods for constructing the sparsity pattern of the resulting ISTL BCRS matrix.

6.5.4 Arbitrary Index Permutations

The order of the variables in a sparse linear problem can have a major impact on the performance of sparse direct solvers [35, 52, 100]; for example, it is important to reduce the matrix bandwidth to minimize the additional fill-in created by a direct solver. The most well-known techniques for this purpose are the Cuthill-McKee algorithm [33] and its reversed variant [53], which are based on reordering rows and columns according to a breadth-first traversal of the system matrix. Another important optimization revolves around reordering the DOFs to create a triangular matrix which can be solved using simple forward or backward substitution, a viable technique for some upwinded FVM schemes for specific transport problems in porous media.

While it is possible to perform the required index permutations in a post-processing step after assembling the algebraic problem, this approach involves copying matrices and vectors and thus incurs a significant overhead in terms of run time and required memory. On the other hand, we can integrate the permutation directly into the index mapping by adding a permutation step to existing nodes in the `Ordering` tree. Such a permutation can be requested by wrapping the original ordering tag in the *decorator tag* `ordering::Permuted` as shown in line 3 of the following example:

```
typedef CompositeGridFunctionSpace<
  VectorBackend,
  ordering::Permuted<InterleavedOrderingTag>,    // permutation decorator
  Children...
```

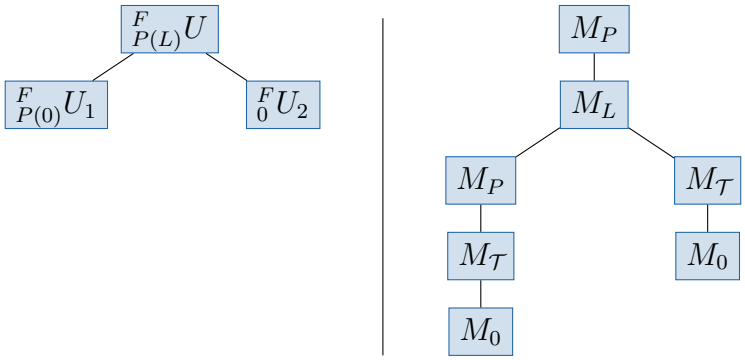


Figure 6.9 — Inserting permutation nodes into an ordering tree. The permutation decorator is denoted by $P(\cdot)$.

```
5     > PermutedSpace;
6   PermutedSpace permuted_space(
7     {{2,1,3}}, // block sizes for the constructor of the nested interleaved tag
8     children...
9     );
```

The resulting ordering tag wraps the original one; it expects an instance of the original tag in its constructor (cf. line 7). After the `Ordering` tree has been initialized, the permutation initially maps every index to itself. It is stored as a `std::vector` inside the ordering tag and can be easily accessed and changed by the user:

```
1   permuted_space.orderingTag().permutation()[i] = j; // access as std::vector
```

At the implementation level, the permutation is performed by an additional *permutation node* M_P which is injected into the ordering tree directly above the node that belongs to the decorated ordering tag. These new nodes differ from those introduced in Section 6.4: they do not correspond to a mapping operation on the original `DOFIndex` tree and thus must not consume an entry of the `DOFIndex` tuple; instead, they only permute the trailing entry of the container index. Given a permutation $P : \mathbb{N} \to \mathbb{N}$, the associated transformation M_P is thus given by

$$M_P\Big((\ldots, i)\Big) = \Big(\ldots, P(i)\Big). \tag{6.9}$$

As an example, in Figure 6.9 some nodes of a function tree have been decorated with a permutation tag; you can see the additional permutation nodes M_P in the associated ordering tree.

Internally, the decorator `ordering::Permuted` is implemented on top of a TMP library that takes care of injecting the additional nodes into the tree and contains infrastructure for hooking into the TypeTree transformation that generates the ordering tree. Using this library, it is also possible to stack multiple decorators on top of each other.

Algorithm 6.1 — Generic index merging for lexicographic and interleaved nodes

```
function GetContainerIndex(Ordering,di,k)
    if Ordering has delegate then
        d ← Ordering.delegate
        ci ← d.DelegateGetContainerIndex(di, k)
    else
        i_c ← di[k]
        co ← Ordering.children[i_c]
        ci ← GetContainerIndex(co, di, k − 1)
        if Ordering sorts lexicographically then                ▷ lexicographic case
            if Ordering creates index blocks then
                ci ← ci ‖ (i_c)
            else
                ci[k] ← ci[k] + Ordering.offsets[i_c]
        else                                                    ▷ interleaved case
            cs ← Ordering.blockSizes[i_c]
            co ← Ordering.blockOffsets[i_c]
            bi ← ci[k] / cs
            bo ← ci[k] (mod cs)
            if Ordering creates index blocks then
                ci[k] ← bo + co
                ci ← ci ‖ (bi)
            else
                bs ← Ordering.blockSize
                ci[k] ← bi · bs + bo + co
    return ci
```

6.5.5 Algorithm

The central task of the `Ordering` tree is the mapping of a given `DOFIndex` $\mathcal{D}$ to its corresponding container index. This operation has to happen for every vector or matrix access in PDELab; its performance is thus very important. On the other hand, designing a well-performing implementation of this mapping algorithm is surprisingly tricky. Most of this difficulty stems from a fundamental mismatch between the template-based `Ordering` tree, which can only be traversed in a way that must be fixed at compile time, and the fact that the indices stored in $\mathcal{D}$ are run time information. One obvious way around this problem is to revert to dynamic polymorphism: If all nodes in the `Ordering` tree inherit from a base class with a virtual function `mapIndex(const DOFIndex&, ContainerIndex&)`, we can look

up the child specified in the `DOFIndex` at run time and dispatch to it via a virtual function call. However, due to our previous experiences with the cost of non-inlined function calls (cf. Table 5.1) and because initial experiments indicated a substantial performance impact of this solution, we instead moved the implementation of the two most common node types (lexicographic and interleaved) into a common base class, reenabling extensive compiler optimizations. This approach results in the recursive Algorithm 6.1. For node types not supported by the default implementation, the algorithm allows an ordering to provide a *delegate* object that uses dynamic dispatch at run time through a virtual function call and is used by the grid entity ordering and the permutation decorator (cf. lines 2 – 5). Note, however, that we avoid the dynamic code path for the common scenario of a scalar function space, where the grid entity is at the root of the ordering tree.

6.6 Impact on Overall Assembly Framework

Several design choices presented in this chapter are unusual when compared with existing PDE assembly frameworks, in particular the idea of a dedicated software component for calculating the DOF order, the introduction of separate data structures for identifying a DOF in a hierarchical discrete function space versus an entry in a LA container as well as the use of multi indices for block matrices and vectors. Gather / scatter operations on sets of DOFs (usually associated with a mesh entity) play a central role in the assembly algorithms of FEM frameworks (cf. Section 2.2.5; consequently, the vastly extended DOF handling architecture we describe necessitates further modifications and optimizations throughout PDELab.

Fortunately, as lined out in [15], PDELab aims at isolating the user as much as possible from directly interacting with the global function spaces and indices; ideally, the user will never have to directly interact with either a `DOFIndex` or a `MultiIndex` object.

On the other hand, we had to modify the framework code in a number of places to retain the performance of the older solution, which was computationally less expensive. In the following, we highlight some important areas that were optimized. With these optimizations in place, the new framework outperforms the old implementation for almost all function spaces; the only exception are very simple layouts like a scalar FVM space; for these special cases, we implemented a very basic alternative ordering infrastructure that can be selected with a different set of ordering tags. The interested reader is referred to the API documentation of PDELab for further details.

6.6.1 Constraints Storage

Function spaces can be constrained for a variety of reasons, including fundamental boundary conditions, process borders in parallel computations and conforming

spaces with non-conforming h-refinement. Typically, the fraction of constrained DOFs in a space will be fairly small; for efficiency reasons, they are therefore stored in sparse data structures. When PDELab was conceived, the only sparse data structure that was readily available in C++ was a `std::map`, which provides a key-value store that is realized as a balanced red-black tree; looking up a key in a map of size N thus requires $\mathcal{O}(\log N)$ element comparisons.

C++11 offers a faster alternative in the form of the hash map implementation `std::unordered_map`. Hash maps offer an average $\mathcal{O}(1)$ lookup time and are thus much better suited to our many-query secenario. At the same time, they have a worst case complexity of $\mathcal{O}(N)$ and rely on a hash function that should generate evenly distributed hash values for all keys.

The objects that we are going to hash are variable-length sequences of non-negative integers (`DOFIndex` and `MultiIndex` objects). For each individual integer, the identity function can serve as a perfect hash (cf. [79]), so that we only need a function $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that combines two hashes. Finding a good function for this purpose proved harder than expected because,apart from the index entry related to the grid entity, the remaining tuple entries are typically all very small integers and thus contain only a very small amount of entropy.

After some failed experiments with simple hash functions from e.g. Boost [114], we realized that a hash table would only be viable with a very strong hash combing function, as those functions would break down when the number of buckets grew beyond ≈ 1000. Fortunately, the growing popularity of interpreted languages and their reliance on hash tables for variable lookup has spawned a number of excellent hash functions that are very well optimized for current central processing units (CPUs). Some of the more prominent examples in this field are FNV [51], SipHash [5, 6] and the CityHash [31] and MurmurHash [90] families of functions. These algorithms are focused on hashing opaque byte streams, usually of substantial length, but CityHash also offers the following implementation for combining two 64 bit hashes:

```
1  std::size_t combine_hashes(std::size_t a, std::size_t b)
2  {
3    const std::size_t kMul = 0x9ddfea08eb382d69ULL;
4    std::size_t c = (a ^ b) * kMul;
5    c ^= (c >> 47);
6    std::size_t d = (b ^ c) * kMul;
7    d ^= (d >> 47);
8    d *= kMul;
9    return d;
10 }
```

This function yields very good bucket distributions even on large composite spaces with up to 1×10^8 DOFs and different matrix ordering and blocking schemes. When considering the full 64 bit out put of this hash function, we were not able to generate a single hash collision between two `DOFIndex` or `ContainerIndex` objects. At the same time, the function is branch-free and avoids expensive operations like

divisions or modulo calculations. On a single core of an Intel Core i7 4960HQ, it is possible to perform about $2.4 \cdot 10^8$ hash combinations per second.

By using `combine_hashes()` to implement a hash computation for our index tuples and subsequently switching the constraints containers and all similar data structures from tree-based maps to hash tables, we were able to completely mitigate the overhead imposed by the more complex indices.

6.6.2 Index Caching and Optimized Batch Mapping

As outlined in the previous sections, we have invested a lot of effort into speeding up the transformation from `DOFIndex` to `ContainerIndex`. Nevertheless, the transformation introduces an additional step into the DOF mapping process, which demands extra computing power and – more importantly – memory bandwidth, the latter one even amplified by the fact that a `DOFIndex` requires a lot more storage than the single integer of a simple, flat index.

On the other hand, most matrix / vector entry access pattern involve multiple associated reads or writes of those values; in the case of matrix assembly, the same indices can even be reused for distinct containers (test and ansatz space vectors and matrix). In order to reduce the performance impact for this important usage scenario, we have implemented a cache that pre-calculates and stores the `ContainerIndices` associated with a `LocalFunctionSpace`. Moreover, mapping all of those values in a single operation makes it possible to use a more efficient algorithm as outlined below, speeding up the transformation process itself.

In the majority of cases algorithms do not require access to individual DOFs; access patterns typically correspond to gather/scatter operations for all values associated with either a specific grid entity or a cell-local function space. Local function spaces are one of the central components in PDELab and are required for virtually all interactions between the user and a discrete function space, while entity-wise access is very rare and currently only used for the entity-based MPI communication facilities of the DUNE grids.

It would of course be possible to simply perform the DOF mapping in those cases by naively looping over the source indices and transforming those one at a time, but such an approach is problematic for performance reasons because it results in a very chaotic memory access pattern and usually requires at least one virtual function call per DOF (cf. Algorithm 6.1). On the other hand, constructing the set of all DOFs for a given cell or grid entity already requires a complete traversal of the function space tree. By reorganizing Algorithm 6.1 to map the complete set of associated DOFs in bulk, it is possible to improve the efficiency of the process and to avoid the virtual function call by combining the per-leaf sizes that are available from the DOF enumeration with the knowledge of the function space structure embedded in the `LocalFunctionSpace` and `Ordering` trees. The resulting algorithm consists of a single traversal of the `Ordering` tree in depth-first order; we implemented it in a

new component called the `LFSIndexCache` and modified all relevant places inside PDELab to use this cache.

In addition to speeding up access to the `ContainerIndices`, the cache also resolves and stores the local constraints structure. PDELab has a very general approach to handling function space constraints: In order to handle situations like hanging nodes refinement, constrained DOFs can create contributions to *arbitrary* other DOFs; in particular, those other entries do not have to be part of the current local function space contained in the `LFSIndexCache`. As described in Section 6.6.1, PDELab stores constraints in a hash map containing the transformed `DOFIndex` values. During matrix assembly, each write to a matrix entry requires looking up the constraints state of the row and column in the constraints maps of the test and ansatz space, respectively. If a constraint is found, the entry is either ignored or accumulated onto different matrix entries as specified in the constraint (the latter happens in the case of hanging nodes constraints). Given local test and ansatz spaces of size N and M, respectively, writing the associated matrix block thus requires $M \times N$ map lookups. By caching the constraints information, the number of lookups is reduced to $M + N$, which yields substantial performance gains in case of higher-order discretization schemes, which have a large number of DOFs per cell.

6.6.3 Optimized Handling of Grid Information

The grid entity transformation (6.3) requires the number of basis functions s_k associated with each grid entity τ_k. There is a large number of grid entities, so storing this information has a significant memory cost; moreover, the frequent lookups greatly increase cache pressure. On the other hand, most FE spaces actually associate the same number of basis function with each grid entity of a fixed type. Consider for example the spaces discussed in Section 2.2.2: A P_1 space for example associates a single basis function with each vertex and no basis functions with any other type of grid entity, while the FV space uses exactly one basis function for each grid cell. If we know that this number is a constant s for all grid entities, we can simplify (6.3) to

$$M_{\mathcal{T}}^{f}\Big((\dots, i), \tau_l\Big) = \Big(\dots, i + ls\Big),$$

and can thus reduce the storage complexity for the grid entity transformation from $\mathcal{O}(|\mathcal{T}|)$ to $\mathcal{O}(1)$. While this equation seems very restrictive, requiring a constant number of basis functions across *all* types of grid entities, remember that, as explained earlier, PDELab actually maintains separate indices and size information for each geometry type, making this optimization applicable to a much larger number of FE bases. The implementation uses a special interface to the basis function definitions to query whether the basis has this constant-size property. If it does, we do not even have to traverse the grid to find out the number of basis functions for each grid entity. If this information is not available a priori (e.g. because the basis supports p-adaptivity), the update process for the ordering

size information traverses the grid and records the per-entity sizes. At the same time, it tracks whether there really are per-entity differences. Each geometry type without any differences is automatically switched back to the constant-size implementation after the grid traversal, freeing the associated per-entity information. As a result, the vast majority of PDELab programs automatically benefit from this optimization.

CHAPTER 7

Implementation Details

In this chapter, we highlight a number of technical details about the implementation of DUNE-MULTIDOMAIN. In general, the software design of these multi domain extensions for PDELAB was driven by two major goals:

Minimal extension of existing API One important goal of the overall design has been an API that reuses the existing PDELAB infrastructure as much as possible, enabling reuse of existing user code (e.g. local operator kernels) and making it easier for PDELAB users to understand and write DUNE-MULTIDOMAIN code.

Exploitation of software modularity to enable code reuse We exploit the existing PDELAB infrastructure to minimize code duplication and to be able to benefit from improvements made to core components. The most important example of this can be found in the support for instationary problems (cf. Section 7.5).

At the highest level, the user interface of DUNE-MULTIDOMAIN is based on a straightforward mapping of the mathematical entities describing the building blocks of a multi domain problem as defined in Chapter 4 to C++ objects. Handling these components in an efficient manner during the performance-sensitive residual and Jacobian assembly process did however pose a number of software design challenges. In the following, we highlight some of those challenges as well as some of our solutions that allow us to reuse large portions of standard PDELAB for functionality that can be used unchanged in a multi domain context.

7.1 Multi Domain Space Composition

In the current implementation, `MultiDomainGridFunctionSpace` is subject to a number of limitations: It only supports lexicographic merging for the DOFs of its direct children, and while it can contain arbitrarily complex composite spaces made up from stock PDELab function spaces, the `MultiDomainGridFunctionSpace` itself has to be at the root of the function space hierarchy, making it impossible to nest those spaces.

Internally, both spaces are based on the same implementation. The only major difference arises from the semantic mismatch between subspaces defined directly on the `MultiDomainGrid` and those that only exist on a `SubDomainGrid`. As explained in Section 3.1.4, each `SubDomainGrid` provides its consecutive numbering of its restricted set of mesh entities. Consequently, a single mesh entity will carry different indices across multiple subdomains. Moreover, the strict static typechecking of C++ makes it impossible for a subspace defined on a `MultiDomainGrid` to operate on `SubDomainGrid` entity objects and vice versa. For this purpose, the TypeTree visitors that traverse the `MultiDomainGridFunctionSpace` use a tag dispatch mechanism (cf. Section 2.4.1) for their callbacks which allows them to differentiate between the different cases; when encountering a subspace defined on a `SubDomainGrid`, the original `MultiDomainGrid` grid cell object is first converted as explained in Section 3.1.4 before it is passed on to the standard PDELab, multi domain agnostic version of the visitor which will then handle the traversal of the current subtree. This conversion process cannot currently be carried out deeper down in the function space tree in a reliable fashion, which explains the restriction on `MultiDomainGridFunctionSpace` nesting.

7.2 Subproblem and Coupling Definition

According to Definition 4.4, a subproblem P is given by a combination of its residual $\mathcal{R}_P$, a predicate function on the underlying mesh and a list of indices $\mathcal{I}_P$ which denote the subspaces referenced in $\mathcal{R}_P$. Given existing C++ objects for the overall multi domain function space, the predicate and the operator (residual), this definition directly carries over to the implementation, as can be seen in the code examples given in Chapter 4.

In addition, we also provide an enhanced version that replaces the numeric indices for the subspaces with their C++ types, improving readability and static error checking by the compiler as well as making the subspace list independent of the actual order of the spaces in the overall multi domain function space. This enhanced version is not available if there are multiple subspaces with identical C++ types, as this precludes the compiler from mapping the subspace type to a unique subspace object.

```
1 // Create subproblem type, alternative version based on subspace types
2 typedef MultiDomain::TypeBasedSubProblem<
```

```
  MultiGFS,
  MultiGFS,
  StokesOperator,
  StokesPredicate,
  // start of subspace list as C++ types for better
  // readability and compiler checking
  StokesVelocityGFS,
  StokesPressureGFS
  > StokesSubProblem;
```

The precise semantics of the predicate object depend on the underlying engine providing the spatial multi domain discretization.

Overall, we are able to provide a very direct translation path between the abstract mathematical objects from our theoretical framework and their corresponding implementation building blocks. Moreover, the resulting C++ objects are very lightweight and carry minimal dependencies – there are no assumptions on any details of the underlying multi domain discretization technique, decoupling the higher-level portions of the simulation setup from the parts responsible for handling the spatial discretization.

7.3 Synthesizing Tailored Subspaces for Residual Components

An important challenge at the implementation level is the integration of existing PDELab local operators for the assembly of subproblem residuals. These implementations expect the local function spaces that they operate on to possess a specific structure with respect to the contained leaf spaces and cannot be used with the entire multi domain function space. On the other hand, all subproblems are required to specify the set of variables that their associated residual is defined on (cf. Definition 4.4). Using this information, we are able to automatically generate a thin wrapper object around the `MultiDomainGridFunctionSpace` that emulates the structure of the function space required by the subproblem and automatically translates any DOF access to the indices defined by the larger `MultiDomainGridFunctionSpace`.

Listing 7.1 — Subproblem with reordered subspaces

```
typedef MultiDomain::SubProblem<
  MultiDomainGridFunctionSpace, // global ansatz space
  MultiDomainGridFunctionSpace, // global test space
  SubProblemLocalOperator,
  Predicate,
  3,1 // subspace index set
  > SubProblem;
SubProblem subproblem(splop,pred);
```

The subproblem shown in Listing 7.1 expects the fourth and the second child space of the overall multi domain function space (indices are zero-based). This example demonstrates two of the central capabilities required from the subproblem-specific subspaces:

Filtering Subproblems are normally not defined on the whole multi domain function space; usually this will not even be possible because not all subspaces will be defined on the spatial domain of the subproblem. The synthesized subproblem subspace thus has to filter out all non-applicable subspaces from the global space.

Reordering As has been explained before, it is highly desirable to support reuse of existing assembly kernels. In general, we cannot assume all those kernels to be written based on a common ordering of the function space components, so the subproblem-specific subspace must support an on-the-fly reordering of the child spaces.

At a lower implementation level, the API of the synthesized subspace depends on the specific set of subspaces: If the subproblem only requires a single subspace, the interface of our synthesized object has to mirror that of the original scalar function space to allow for transparent usage. On the other hand, the selection of multiple subspaces requires the on-the-fly creation of a new `LocalFunctionSpace` node to aggregate the selected set of subspaces. In our implementation, this synthesized node always mirrors the inteface of a standard PDELab `CompositeGridFunctionSpace`.

The creation of these synthesized spaces is mostly enabled by several pieces of support infrastructure from the TypeTree library, in particular proxy nodes (Section 5.7.1) for the single-component case and filtered nodes (Section 5.7.2) if the subproblem operates on multiple child spaces.

By only ever constructing these subproblem-specific subspaces in a local fashion, we minimize the associated overhead, as there is only a small number of basis function associated with each grid cell for which we have to store mapping information between the overall space and the subspace. Moreover, the number of local basis functions does not depend on the overall size of the global space across the grid, which avoids potential scalability issues for large simulations.

7.4 Efficient Creation of Statically Typed Visitor Captures

During assembly, the infrastructure code has to iterate over the list of subproblems and couplings (called assembly participants or just participants in the following) and invoke their respective assembly kernels on the current grid cell / intersection. As the participants are stored in a TypeTree tree of depth 1, this iteration is

Datum	Kernel type				
	Cell	Boundary	Skeleton	Coupling	Mortar Coupling
Cell	yes				
Intersection		yes	yes	yes	yes
Interior spaces	yes	yes	yes	yes	yes
Interior solution	α	α	α	α	α
Interior jacobian	yes	yes	yes	yes	yes
Exterior spaces			yes	yes	yes
Exterior solution			α	α	α
Exterior jacobian			yes	yes	yes
int $\rightarrow$ ext jacobian			yes	yes	
ext $\rightarrow$ int jacobian			yes	yes	
Coupling spaces					yes
Coupling solution					α
Coupling jacobian				yes	yes
int $\rightarrow$ cpl jacobian					yes
cpl $\rightarrow$ int jacobian					yes
ext $\rightarrow$ cpl jacobian					yes
cpl $\rightarrow$ ext jacobian					yes

Table 7.1 — Capture contents for jacobian visitors. α denotes a type of information that is only required for the solution-dependent kernels like α^{vol} etc.

performed by applying a visitor to that tree. Depending on the current assembly operation (residual or Jacobian) and the type of kernel (volume, boundary, coupling, etc.), those participant-specific kernels need access to different amounts and types of information. Table 7.4 contains a list of the different kinds of information required by the kernels invoked for Jacobian assembly. All of the objects containing this information have to be stored inside the visitor traversing the assembly participants together with precise type information that must be made accessible to the assembly participants.

Looking at Table 7.4, we can see that this requires a substantial number of different visitor implementations, which all differ only by the type of stored information. Keep in mind that it is not possible to instead use a single implementation that contains all data, as that data will not be available in all contexts. For example, when calculating a cell integral, it is not possible to store an intersection in the visitor.

In order to avoid reimplementing the data storage part of the visitor over and over again, we have devised a scheme that automatically constructs a custom visitor from information passed to it at construction time and that retains static information about all stored types. We achieve this by defining pairs of data

Listing 7.2 — Data wrapper for a cell object

```
template<typename Cell_>
struct cell_wrapper
{
  typedef Cell_ Cell;

  cell_wrapper(const Cell& cell)
    : _cell(cell)
  {}

  // prevent copying, but allow moving
  cell_wrapper(const cell_wrapper&) = delete;
  cell_wrapper& operator=(const cell_wrapper&) = delete;

  const Cell& cell() const
  {
    return _cell;
  }

private:

  const Cell& _cell;
};

template<typename Cell>
cell_wrapper<cell> wrap_cell(const Cell& cell)
{
  return cell_wrapper<Cell>(cell);
}
```

containers and container generation functions for all required types of objects. An example, used to capture a grid cell, is shown in Listing 7.2.

The actual implementation uses several variations of these wrappers, which are geared towards different capture scenarios, e.g. capture by reference vs. capture by value and read-only vs. writable captures. As there are only a small number of these variants, we are able to avoid massive code duplication by generating all required wrappers by means of a small number of preprocessor macros.

In a second step we then pass these wrapper objects to a generator for the actual visitor, which inherits from all of them, in turn making their accessor **`typedef`**s and methods available inside the visitor. At the same time, this technique avoids any accidental double passing of data, as the compiler will generate an error due to the overload ambiguity introduced by having the same method / type name available in multiple base classes. In order to avoid correctness and performance problems, the wrappers are designed to be non-copyable, so the construction process relies on C++11 perfect forwarding and move construction to initialize the visitor bases which store the captured data. Moreover, the implementation also requires on variadic templates to support an arbitrary number of data wrappers.

Listing 7.3 — Functor for invoking participant-specific local operator method

```
template<typename Data>
struct alpha_volume
  : public data_accessor<Data>
{

  using data_accessor<Data>::data;

  template<typename SubProblem>
  void operator()(const SubProblem& subProblem)
  {
    if (!subProblem.appliesTo(data().eg()))
      return;
    Data::Operator::extract(subProblem).alpha_volume(
      data().eg(),                  // grid entity
      LFS::lfsu(data(),subProblem), // synthesize subproblem subspace
      data().x(),                   // local solution DOFs
      LFS::lfsv(data(),subProblem), // synthesize subproblem subspace
      data().r()                    // local residual DOFs
      );
  }

};
```

The actual visitor is then built by parameterizing the generic `visitor` template with two meta functions that select the actual function to be called and subsequently calling a static member function on the parameterized visitor, feeding it the result of the `wrap_*()` calls introduced above. In the case of the `alpha_volume()` method, the visitor is created like this:

```
// specify kernel (alpha_volume) and corresponding call guard
typedef visitor<functors::alpha_volume,do_alpha_volume<> > Visitor;
// start subproblem traversal with temporary visitor object
localAssembler().applyToSubProblems(
  // generate visitor by specifying capture data
  Visitor::add_data(
    wrap_operator_type(DefaultOperator()),
    wrap_eg(eg),
    wrap_lfsu_s_cache(lfsu_s_cache),wrap_lfsv_s_cache(lfsv_s_cache),
    wrap_x(data().x_s),wrap_r(data().r_s_view)
  )
);
```

The visitor returned by `add_data()` implements the TypeTree visitor interface and is then applied to the tree of assembly participants by the local assembler. It invokes the `functors::alpha_volume` functor for each participant, which unpacks the data stored in the visitor and calls the `alpha_volume()` on the local operator with the unpacked information (cf. Listing 7.3).

7.5 Time Dependent Problems

PDELab features an efficient implementation of time integrators based on the method of lines. As we have seen in Section 2.1.5, these time integrators require separate access to the temporal and the spatial parts of the residual.

In order to avoid a complete reimplementation of the program logic associated with the assembly and correctly weighted accumulation of those individual residual and Jacobian contributions, PDELab contains a callback-based generic assembler for one-step time integration methods. This assembler uses two separate stationary `GridOperator`s for the spatial and the temporal contributions of the residual and the Jacobian, respectively (cf. Section 2.1.5. It works by extracting the appropriate callback engines (for pattern assembly, residual or jacobian assembly) from those two stationary assemblers and wrapping them in a new callback engine that performs the necessary evaluations of both components at different timepoints, accumulating their results and weighting them as required by the time integration scheme. These callback engines only operate at the level of a single grid cell and thus do not have to be aware of the overall grid structure. In order to traverse the grid (or multiple grids in our multi domain setting), the stationary grid operators provide a special component called the *global assembler*. This component accepts a callback engine and traverses the grid, setting up the required local function spaces, handling the global vectors and matrices and invoking the appropriate callback functions of the callback engine for every cell and intersection with only local information. The instationary grid operator then uses the global assembler of one of the stationary operators to apply its newly constructed callback engine to the grid without requiring any further knowledge about how to perform the grid traversal and / or DOF handling.

This separation of concerns at the stock PDELab level makes it possible to reuse the default `OneStepGridOperator` to assemble instationary multi domain problems, which has the double advantage of avoiding code duplication inside our multi domain extensions and providing users with a uniform approach to the assembly of instationary problems, irrespective of the spatial problem structure.

7.6 Performance Characteristics

In our implementation, we exploit the fact that the entire multi domain problem is always defined on a single `MultiDomainGrid`. We can thus perform the problem assembly by iterating over that common, underlying grid. We then extend the existing PDELab assembler with additional logic that checks the current integration domain (in our case, cells and cell intersections) for membership in each of the defined subproblems and couplings by evaluating their associated predicate, which introduces an overhead of $\mathcal{O}(N_S + N_C)$, where N_S and N_C denote the number of subproblem and coupling components.. On the other hand, the evaluation of those

predicates will normally be very cheap, as it only involves the comparison of two sets of `SubDomainGrid` indices.

Another, more substantial source of overhead is the construction of the subproblem-specific subspaces, which requires setting up a mapping between the local basis function indices in the overall local function space and the subproblem-specific subspace, and the performance penalty related to the use of meta grids and the required conversions of grid objects between those meta grids.

As we will see in the next Chapter, using Dune-Multidomain does impose a certain performance penalty relative to a vanilla PDELab simulation, but as those numbers will show, its extent is in our opinion entirely acceptable, as we also have to take into account that the actual problem that we are solving is more complicated.

CHAPTER 8

Applications

The creation of a software framework like the one presented in this thesis involves a major amount of effort, but taken by itself, the result is not overly useful. To become a success, it must be used to build real applications, and implementing those applications should become substantially easier with its help.

Dune-Multidomaingrid passes both of these hurdles: In the first part of this chapter, we show how the framework helps in building simulations for two of the example problems from Chapter 4, allowing us to reuse a lot of existing building blocks and minimizing the amount of new code. Moreover, we investigate the performance of the framework for the coupled Poisson problem by comparing it to an equivalent single domain problem simulation using standard PDELab.

Since its early development phase, our framework has been used as a tool by other researchers building multi domain applications. In the second part of the chapter, we present a short overview of a doctoral dissertation that heavily relied on Dune-Multidomain for the numerical studies.

Note that in addition to this example, there is a substantial number of external research projects based on Dune-Multidomain, including simulations of matter transport through capillary blood vessel walls [8, 7], of a coupled system of two-phase porous medium and one-phase free flow with evaporation [87, 88] and of solid-state laser resonators [124]. Moreover, there is an ongoing PhD project based on the coupled two-phase / one-phase porous medium model presented in Section 4.2.3.

8.1 Poisson Problem on Two Subdomains

We begin by looking at the simple model example (4.2) from Chapter 4, which describes two coupled Poisson problems on the left and right halves of the unit square:

$$\begin{aligned} -\Delta u_i &= f && \text{in } \Omega_i, \quad i \in \{L, R\}, \\ u_i &= g && \text{on } \Gamma_{i,D}, \\ \nabla u_i \cdot \boldsymbol{n} &= j && \text{on } \Gamma_{i,N}, \\ u_L &= u_R && \text{on } \Gamma_C, \\ (\nabla u_L) \cdot \boldsymbol{n} &= (\nabla U_R) \cdot \boldsymbol{n} && \text{on } \Gamma_C. \end{aligned}$$

This version of the problem is slightly more general in that it allows for both Dirichlet and Neumann boundary conditions. This example can also be treated as a "normal" PDE problem with a single variable u on the entire domain Ω, which we have used to measure the overhead of our framework relative to a standard implementation in stock PDELab.

The domain was discretized with the structured mesh `YaspGrid` contained in Dune-Grid by refining an initial grid with a single cell on the unit square. After refinement, the grid was wrapped in a `MultiDomainGrid` and the left and the right half of the domain were assigned to different subdomains Ω_L and Ω_R. In order to demonstrate the suitability of our framework for both conforming and non-conforming discretizations, we then used several different continuous and discontinuous discretizations for the subproblems on those domains.

Before looking at an actual multi domain problem, we investigated the performance overhead imposed by the additional infrastructure of the multi domain assembler by comparing the assembly performance of stock PDELab and our extensions for a single domain problem. To do so, we first created a standard PDELab function space and `GridOperator` directly on top of the `MultiDomainGrid` and measured the run time of the most important assembly operations, in particular constraints evaluation, matrix pattern construction and residual and Jacobian evaluation.

Afterwards, we created a `MultiDomainGridFunctionSpace` with a single child space for the whole domain Ω and used it to define one `SubProblem` combining the single child space with the local operator from the stock PDELab version. By feeding that subproblem to a multi domain `GridOperator`, we can replicate the standard PDELab setup without subdomain extensions.

We then measured the performance of both versions on a grid with 262 144 cells using Q_1 and Q_2 spaces and conforming as well as non-conforming discretizations. The results are shown in Table 8.1. In most cases, going from the default PDELab version to the multi domain aware assembler increases run time by about 10%, which in our view is reasonable given that the framework needs to perform additional subdomain memberships checks for every mesh cell and intersection. As we have seen in Chapter 3, the overhead of these checks is especially pronounced for fast, structured host grids like the `YaspGrid` used in this example.

Discretization	Mode	Constraints	Pattern	Residual	Jacobian
P1	basic	0.18	0.14	2.11	1.30
P1	wrapped	0.20	0.15	2.26	1.30
P1 - P1	MD flat	0.49	1.04	3.20	2.50
P1 - P1	MD blocked	0.51	1.06	3.19	2.58
P2	basic	0.22	3.47	3.56	2.53
P2	wrapped	0.27	3.39	3.82	2.59
P2 - P2	MD flat	0.72	4.77	5.11	4.27
P2 - P2	MD blocked	0.71	4.97	5.21	4.56
DG1	basic	0.18	0.93	1.45	1.71
DG1	wrapped	0.19	1.02	1.69	1.96
DG1 - DG1	MD flat	0.50	1.77	2.34	2.63
DG1 - DG1	MD blocked	0.51	1.81	2.30	2.77
DG2	basic	0.21	4.66	2.36	5.34
DG2	wrapped	0.26	4.76	2.67	5.47
DG2 - DG2	MD flat	0.71	5.86	3.63	6.50
DG2 - DG2	MD blocked	0.73	6.62	3.61	7.16

Table 8.1 — Runtime of common assembly operations for Poisson problem. All timings are in seconds. Benchmark performed on hardware configuration B.2. *basic* denotes a problem run directly on the host grid, *wrapped* one run on a single large subdomain, and the remaining two signify a multidomain problem with the two spaces listed on the left and either a flat vector backend or one with one large block for each subdomain.

Another important question is that of how much single physics code we were able to reuse for the multi domain simulation. This example was created by modifying an existing PDELab program and simply replacing the standard PDELab components with their multi domain aware counterparts as introduced in Chapter 4, which is a very straightforward process for a developer with a decent knowledge of PDELab. The only major changes involved setting up the subdomain layout in the `MultiDomainGrid`, duplicating the VTK output code to write separate files for each subdomain and, most importantly, the implementation of the local coupling operator, which you can find in Listing A.1. In total, the relevant changes to the problem (including the new operator) amounted to about 200 lines of code, which are moreover easily understandable by a developer with prior experience in standard PDELab. On the other hand, we were able to reuse existing implementations for a continuous Galerkin discretization of the Poisson problem as well as an advanced SIPG scheme for the DG discretization that would have required a major amount of effort to reimplement.

8.1.1 Iterative Solution With Dirichlet-Neumann Scheme

In order to demonstrate our support for weakly coupled solution schemes, we have also implemented a version of the coupled Poisson-Poisson problem that uses a Dirichlet-Neumann fixed point iteration that alternates between solving the problem on each subdomain in lockstep, using the solution on the other subdomain to provide either Dirichlet or Neumann boundary data on the coupling interface. A detailed mathematical description of this scheme can be found in [123]. In the following, we provide an outline of how this scheme can be implemented in Dune-Multidomain to demonstrate the large amount of support our framework provides for the development of advanced multi domain solution.

As a starting point, we assume that the simulation with a monolithic solver has already been set up as explained in the previous section. We will reuse the function spaces and subproblems created for that problem. However, we have to use a different vector backend tag for the `MultiDomainGridFunctionSpace`:

```
typedef Dune::PDELab::ISTLVectorBackend<
  Dune::PDELab::ISTLParameters::dynamic_blocking
  > MDVBE;
```

With this tag, PDELab will create vectors and matrices with separate blocks for each subproblem:

$$\begin{pmatrix} A_L & 0 \\ 0 & A_R \end{pmatrix} \begin{pmatrix} u_L \\ u_R \end{pmatrix} = \begin{pmatrix} f_L \\ f_R \end{pmatrix}.$$

As our problem only couples via boundary conditions (i.e. the right hand side), the off-diagonal blocks of the matrix are empty. With this matrix structure in place, we can directly access the individual blocks and for example pass A_L to an ISTL solver without having to copy any data. At the same time, Dune-Multidomain retains all information about the global problem structure and identifies corresponding DOFs on each side of the coupling interface.

As a first step, we need to create a local operator that calculates the residual contributions of the Neumann and Dirichlet coupling terms. For the purpose of this example, we have decided to apply both Neumann and Dirichlet boundary conditions on the coupling interface in a weak fashion; the implementation of the local operator can be found in Listing A.3. It is based on the default PDELab DG operator for convection-diffusion-advection problems and was created by changing the skeleton integrals to coupling terms and adjusting them to assemble either Neumann or Dirichlet boundary conditions using the solution from the other subdomain as boundary data. All other integration kernels were simply deleted. We create two instances of this operator, one for each subdomain:

```
typedef ConvectionDiffusionDGNeumannDirichletCoupling<
  Params
  > NDCoupling;                    // type of the coupling operator
NDCoupling nd_coupling_neumann(    // variant that applies a Neumann coupling
  CouplingMode::Neumann,
  params,
```

```
  ...
  );
NDCoupling nd_coupling_dirichlet(  // variant that applies a Dirichlet coupling
  CouplingMode::Dirichlet,
  params,
  ...
  );
```

With these local operators in hand, we can then create two coupling descriptors. In this case, we decide to couple the left subproblem to the right subproblem by means of Dirichlet conditions:

```
typedef MultiDomain::Coupling<
  LeftSubProblem,
  RightSubProblem,
  NDCoupling
  > DirichletCoupling;
DirichletCoupling dirichlet_coupling(
  left_sp,
  right_sp,
  nd_coupling_dirichlet
  );
```

Here, we omit the second coupling; it is an exact mirror of the first one, apart from the different local operator for the coupling (Neumann instead of Dirichlet).

Before we can create a `GridOperator`, we first have to construct a matrix backend that can store an individual number for the count of nonzero entries per row of each matrix block:

```
typedef std::array<std::array<std::size_t,2>,2> EntriesPerRow;
typedef istl::BCRSMatrixBackend<EntriesPerRow> MBE;
MBE mbe(
  {{
    {27,  0},  // number of nonzero entries for each
    { 0, 27}   // of the four large matrix blocks
  }});
```

For the Dirichlet-Neumann scheme we then need two separate `GridOperator`s, one for each subdomain. Consequently, each operator assembles one subproblem and the associated coupling descriptor:

```
typedef MultiDomain::GridOperator<
  MultiGFS,MultiGFS,
  MBE,R,R,R,C,C,
  LeftSubProblem,
  DirichletCoupling
  > LeftOperator;
LeftOperator left_operator(
  multigfs,multigfs,
  cg,cg,
  mbe,
  left_sp,
  dirichlet_coupling
  );
```

We again omit the code for the second `GridOperator`. Each of these two grid operators only assembles contributions into the parts of the residual and the Jacobian matrix that belong to its corresponding subproblem.

In order to complete our simulation, we now need to set up a linear solver backend for the problem. For this purpose, we use the custom `ISTL_SEQ_Subblock_Backend`, which accepts the global system matrix containing both subproblems, but only operates on one block row of the matrix, corresponding to one subproblem:

```
typedef ISTL_SEQ_Subblock_Backend<
  SeqSSOR,
  BiCGSTABSolver
  > LinearSolver_DN;
LinearSolver_DN linear_solver_dn_0(
  0,  // select the block row on which to operate (0 for left, 1 for right)
  ... // solver parameters
  );
```

Finally, we can create a pair of standard PDELab linear problem solvers, which take care of assembling residuals and Jacobians and of solving the assembled system by running the linear solver backend:

```
typedef Dune::PDELab::StationaryLinearProblemSolver<
  LeftOperator,
  LinearSolver_DN,
  V
  > LeftPDESolver;
LeftPDESolver left_pde_solver(
  left_operator,      // GridOperator for left subproblem
  linear_solver_dn_0, // linear solver for corresponding matrix block
  ...                 // solver parameters
  );
```

With all components in place, we are only left with implementing the actual fixed point iteration, which now becomes a very short loop that directly corresponds to the mathematical description of the scheme:

```
while (r / r_0 > rel_reduction && r > max_error && i < max_iterations)
  {
    u_old = u;
    left_pde_solver.apply(u,!reassemble_jacobian_left && (i>0));
    u *= alpha;
    u.axpy(1.0-alpha,u_old);

    u_old = u;
    right_pde_solver.apply(u,!reassemble_jacobian_right && (i>0));
    u *= alpha;
    u.axpy(1.0-alpha,u_old);

    r = residual.two_norm();
    ++i;
  }
```

Solver	h	inner reduction	dampening	iterations	time (s)
Monolithic	2^{-7}	–	–	204	4.87
Neumann-Dirichlet	2^{-7}	10^{-4}	0.8	67	202
Dirichlet-Neumann	2^{-7}	10^{-4}	0.8	24	52.4
Dirichlet-Neumann	2^{-7}	10^{-4}	0.7	29	62.5
Monolithic	2^{-6}	–	–	105	0.559
Dirichlet-Neumann	2^{-6}	10^{-4}	0.8	17	8.07

Table 8.2 — Iteration counts and solution times of monolithic solver and Dirichlet-Neumann iteration for coupled Poisson problems. The solution time comprises the complete simulation (assembly and solver). Benchmark performed on hardware configuration B.2.

Here, we have assumed that a suitable start defect r_0 has been calculated outside of the loop; $\alpha \in (0, 1)$ is a damping parameter. A fully working implementation of this example can be found in the test directory of Dune-Multidomaingrid.

Similar to the monolithic solver, we were able to reuse a large number of building blocks for this implementation; the subproblems and their local operators did not have to be changed from the monolithic multi domain problem. At the same time, we were able to completely change the block structure of the linear algebra containers by simply changing a single template parameter on the otherwise unchanged function space. Finally, while this problem involves more grid operators and solvers, they directly stem from the fact that we need to solve two separate problems for the Dirichlet-Neumann iteration; furthermore, the construction of all these components is very straightforward as long as the user takes care to use the special linear solver backend. The only really new code for this example was the local operator for the coupling, and even here we managed to profit from the fact that the interface of that operator in Dune-Multidomain closely resembles a standard DG skeleton residual, allowing us to copy and paste most of the code from an existing DG operator implementation.

We have assessed the performance of this weakly coupled solver by comparing it with that of the monolithic solver from the previous section. We chose a combination of a continuous Q_2 space and a DG_2 space to demonstrate the versatility of our Dirichlet-Neumann implementation, which applies the coupling boundary conditions at the equation level instead of the linear algebra level. We solved the problem with a target defect reduction of 10^{-10}; the Dirichlet-Neumann solver was run with different damping parameters for the iterative scheme and we reduced the precision of the subdomain solves to 10^{-4} to improve its performance. The results can be found in Table 8.2. As is to be expected in such a simple setting, the monolithic solver greatly outperforms the iterative scheme because it has a much more information about the coupling effects between the two subdomains. Keep in

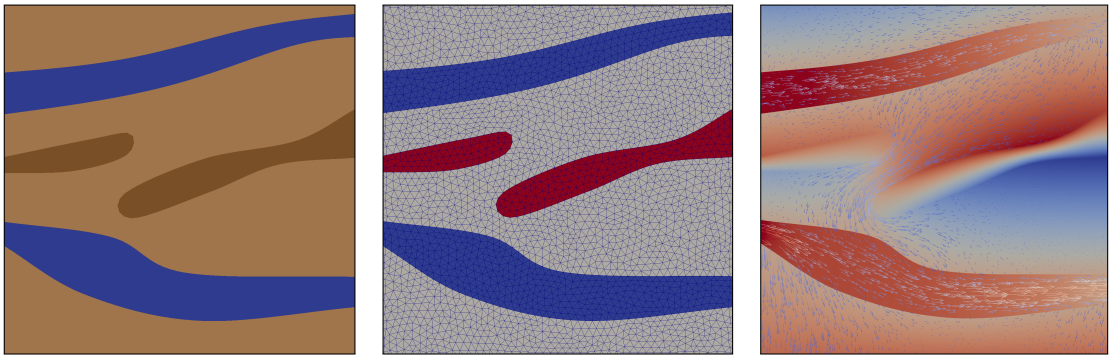


Figure 8.1 — Stokes-Darcy problem solved using our example application built using DUNE-MULTIDOMAIN. Left: Domain with two free-flow channels from left two right and two impermeable areas in the porous medium. Center: Computational grid. Right: Pressure and velocity fields of an example simulation.

mind that the Dirichlet-Neumann scheme only lists the number of outer iterations; each of those iterations involved two linear solves, one for each subdomain, albeit with less precision than the linear solver in the monolithic approach.

Looking at this example, we conclude that our framework has managed to attain its goals of greatly simplifying the development of multi domain simulations, while at the same time offering the developer flexibility in the solution scheme and managing to provide competitive performance.

8.2 Coupling of Free Flow and Porous Media

While the coupled Poisson problem can serve as a good validation example for the performance and implementation impact of the framework, it is not a real multi domain problem for which the framework will be used by application scientists. For this reason, we have also implemented a simulation of the Stokes-Darcy problem described in Section 4.2.2.

Coupled systems of groundwater flow in porous media and embedded free-flow channels are important for a good understanding of karst aquifers. Karst aquifers are a common source of drinking water and consequently, extensive knowledge of the transport processes within those reservoirs is of vital importance for ensuring water quality, e.g. to accurately predict the spreading of pollutants, making this example much more relevant to real-world applications.

For our experiments we used the formulation laid out in equations (4.6) – (4.8), which is based on the approaches in [29, 49]. We discretized the free flow domain using a standard Taylor-Hood element (P_2/P_1) and employed a SIPG scheme with 3rd order elements in the porous medium. Note that we refrain from a mixed formulation in the porous medium, thus creating a problem that couples

two domains with genuinely different variables. Earlier examples of employing Discontinuous Galerkin methods for at least one of the two domains can be found in [110, 55, 54, 76]. A comprehensive comparison of different combinations of CG and DG discretizations in the respective subdomains can be found in Chidyagwai and Rivière [30]. Figure 8.1 shows an example simulation performed with our code.

Note that for the purpose of this thesis, we were mostly interested in the Stokes-Darcy problem from the point of view of system assembly. For that reason, we restricted ourselves to solving the resulting linear algebraic problem with the direct solver library SuperLU [35]. On the other hand, there are very interesting domain decomposition approaches to solving this type of problem, e.g. [37, 38, 9], where the authors propose a nested iteration on the outer and inner degrees of freedom of the two subdomains using a Steklov-Poincaré type operator. Implementing this approach and other advanced solvers might serve as a good validation platform for future additions to the library at the solver level.

As in the Poisson example, we did not have to create the local operators for the individual subproblems from scratch, as PDELab already contained appropriate implementations that we were able to fit into the multi domain setting with minimal effort. As a result, we only had to write an operator for the Beavers-Joseph-Saffman conditions (4.8), which can be found in Listing Listing A.2 and contains 185 lines of code, and the problem setup and driver routine, which comprises about 240 lines of code, including grid setup and solution output. The complete source code for this example can be found in the test directory of Dune-Multidomain, in the file `stokesdarcy2.cc`.

A much more elaborate work on a coupled system of two-phase porous medium and one-phase free flow with evaporation that was implemented on top of Dune-Multidomain and the DUNE-based porous media framework DuMu^x [50] can be found in [87, 88].

8.3 Signal Transport in Neurons

Detailed biophysical models and simulations are an important tool for increasing our understanding of complex biological processes and for validating existing, more coarse-grained functional models of those processes. Steady increases in computing capacities and advances at the model development level have been driving a trend towards more complex simulations that couple multiple physical processes; at the same time, biological simulations often involve complex geometries. Taken together, these challenges form a perfect application scenario for our multi domain software framework. In the following, we present how Dune-Multidomain was successfully used to overcome those technical challenges in a recent simulation of neuron signal propagation by Pods, Schönke, and Bastian [105] and Pods [104].

Their work focuses on the propagation of action potentials along axons (a kind of fiber connecting two neurons that acts as a communication channel). The

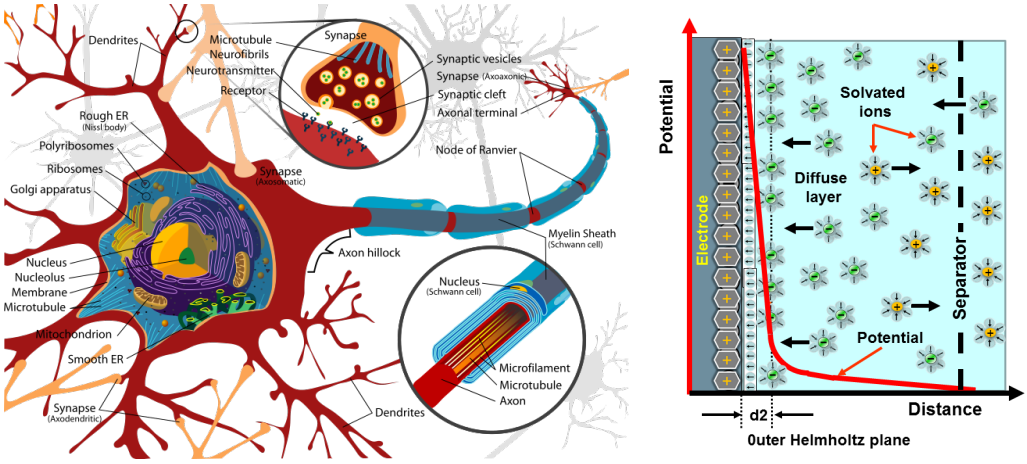


Figure 8.2 — Left: Structure of a neuron and an axon. By LadyofHats, via Wikimedia Commons[Public Domain]. Right: Schematic picture of a membrane and electrolyte. By Elcap, via Wikimedia Commons[CC0].

general situation is depicted in Figure 8.2: A signal is initiated in one neuron and then travels along the axon to a second neuron, where it triggers a signal by releasing special neurotransmitter molecules. The axon is a hollow, fluid-filled fiber embedded in the extracellular fluid. Its hull is a membrane that separates the charges created by the ion solutions on either side of it, but also contains special ion channels that will allow ions to traverse the membrane under certain circumstances. The signal transmitted along the axon takes on the form of a voltage pulse; it travels along the axon by a complex interaction between the electrostatic potential, the ion concentrations on either side of the membrane and the different activation mechanisms of the membrane ion channels. Of particular interest for these interactions is a thin layer in direct vicinity to the membrane called the *Debye layer* [86], which exhibits a strong gradient in both the potential and the ion concentrations (see Figure 8.2) and has to be resolved at a very high resolution in the normal direction of the membrane interface.

In the model employed by the authors, the evolution of the ion concentrations and the resulting electric potential is described by the Poisson-Nernst-Planck equations of electrodiffusion [84], which model the ion movement using the Nernst-Planck equation

$$\frac{\partial n_i}{\partial t} + \nabla \cdot \boldsymbol{F}_i = 0,$$

where $\boldsymbol{F}_i$ denotes the ion flux of ion species i and is defined by

$$\boldsymbol{F}_i = -\boldsymbol{D}_i(\nabla n_i + z_i n_i \nabla \phi).$$

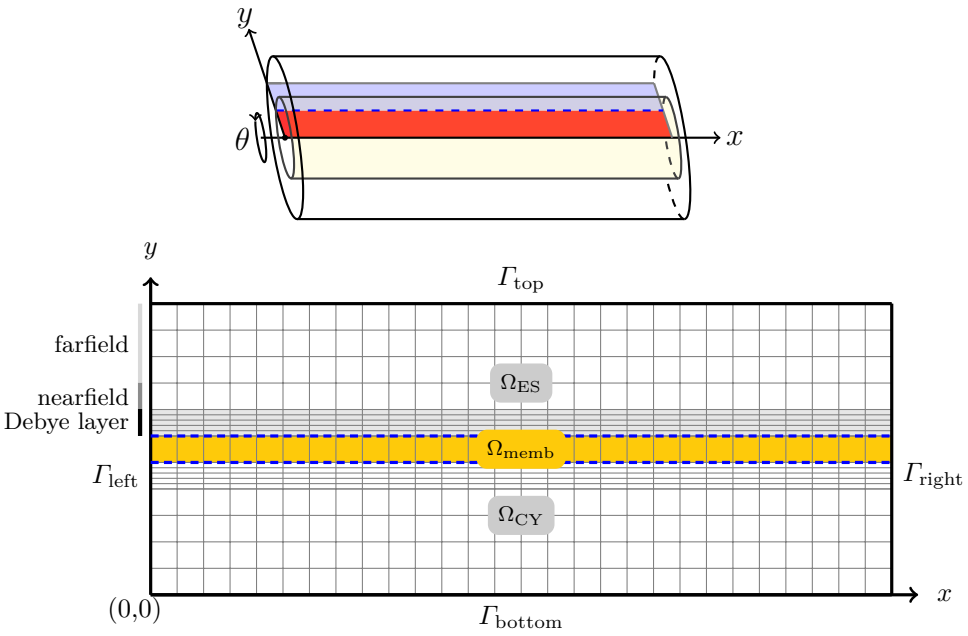


Figure 8.3 — Two-dimensional computational domain for the cylinder-symmetric axon model. As can be seen in the upper part, the authors regard a 3D model, but use cylinder symmetry to perform the computation on the 2D mesh shown in the lower part. Their domain is divided into two subdomains, one (non-connected) electrolyte domain $\Omega_{\text{elec}} = \Omega_{\text{CY}} \cup \Omega_{\text{ES}}$ and the membrane subdomain Ω_{memb}. The most interesting part of the domain is the Debye layer close to the membrane, which is resolved with a much higher resolution than the remainder of the domain. – Image courtesy Jurgis Pods.

These fluxes are in turn coupled with the dimensionless electric potential ϕ by the ion concentrations. The potential itself is described by the Poisson-type equation

$$\nabla \cdot (\epsilon \nabla \phi) = -\frac{e^2 n^*}{\epsilon_0 k_B T} \sum_i z_i n_i.$$

Here, n_i are relative concentrations of the ion concentrations with respect to the scaling concentration $n^* = N_A$ (the Avogadro constant), while z_i denotes the valence of ion species i and D_i its position-dependent diffusion coefficient; T is the temperature, k_B the Boltzmann constant and ϵ the relative permittivity, which may also depend on x.

These equations are coupled to a system of ODEs describing the ion channel activity based on the Hodgkin-Huxley model [69].

8.3.1 General Simulation Setup

As depicted in Figure 8.3, the simulation domain consists of two different subdomains: the intra- / extracellular fluid with ion concentrations and electric potential (this is an example of a subdomain which consists of multiple non-connected parts) and the cell membrane, which is modeled as a very thin (thickness 1 mesh cell) layer that only carries the electric potential. The ion concentrations on the two sides of the membrane are coupled by means of Neumann-type boundary conditions derived from the ODE system of the Hodgkin-Huxley model.

Both electrostatic potential and the ion concentrations are discretized with standard Q_1 finite elements and the resulting nonlinear system is – apart from the ODEs coupling the two sides of the membrame – solved in a fully coupled fashion using an implicit Euler timestepping scheme and a Newton solver. The actual implementation of this scheme is rather involved, however: It starts with an instance of the structured grid implementation `YaspGrid`, which is wrapped in a meta grid that replaces the uniform geometry of the original grid with a tensor product geometry that provides a very fine resolution in normal direction to the membrane within the Debye layer. This grid is then again wrapped inside a `MultiDomainGrid` that divides the mesh into the two subdomains of the problem.

For the local operators, the authors were able to reuse the existing generic `Dune::PDELab::ConvectionDiffusionFEM` implementation for the potential equation on the membrane. The associated subproblem was set up in such a way as to only present the subspace with the electric potential to that operator. The authors then developed an extended version of this generic operator that operates on the full set of variables (potential and ion concentrations) and which incorporates the additional residual of the Nernst-Planck equations.

These local operators were encapsulated in appropriate `SubProblem`s. The main area that lacked extensive support at the framework level was the implementation of the coupling boundary conditions with their embedded ODE solver; this part of the simulation required a lot of implementation effort by the authors. At the same time, they point out that this type of coupling is a very special case that will be difficult to support in a generic fashion.

The `GridOperator`s for the spatial and temporal part of the residual are regular `MultiDomain::GridOperator`s, while the overall assembler for the instationary problem (`OneStepGridOperator`), the time integrator and the solver for the algebraic problem that occurs for each time step have been constructed from standard PDELab components, again demonstrating the large amount of code reuse facilitated by our library and the abstractions in the underlying PDELab toolkit. As our multi domain extensions reuse the time integration infrastructure and the linear solvers from PDELab, the authors did not have to learn a second set of APIs to implement functionality that is essentially unchanged from a single domain problem.

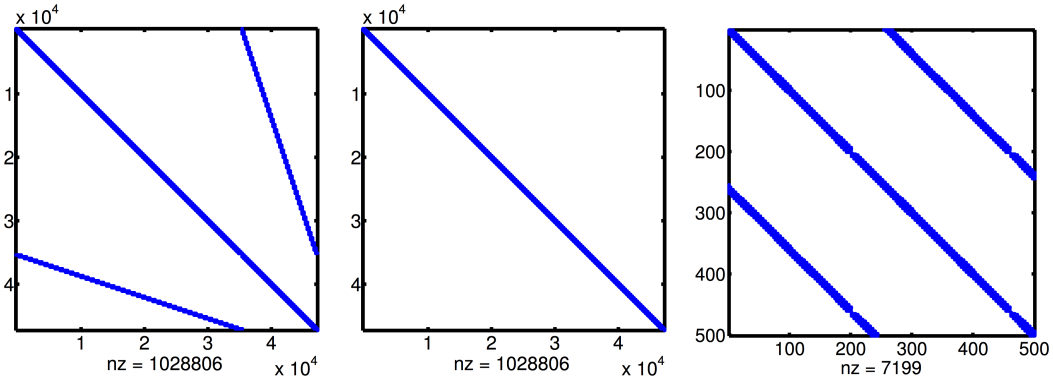


Figure 8.4 — Comparison of Matrix structure for neuron transport problem depending on DOF ordering. Left: Default (lexicographic) order with unfavorable sparsity pattern. Center: Entity-blocked ordering with improved matrix structure. Right: Zoomed view of the entity-blocked ordering, revealing a tri-band structure typical of scalar 2D problems on structured meshes. – Image courtesy Jurgis Pods.

The authors initially fought with the stability and performance of the linear solver embedded inside the Newton solver. They were able to resolve this problem by globally grouping the DOFs from all subspaces by entity (creating blocks with the structure $(\phi, n_1, \ldots, n_N)$) and mirroring the resulting block structure in their vectors and matrices. This change greatly improved the structure of their system matrix (cf. Figure 8.4) and, importantly, allowed them to use block-aware versions of ILU or AMG as preconditioners for their linear solver. These preconditioners then performed an exact inversion of the small diagonal matrix blocks attached to each grid vertex, greatly improving convergence rates and enabling the authors to scale their numerical experiments to much larger problem sizes.

8.3.2 Parallel Computations

While we have omitted a discussion of this feature in the present work, DUNE-MULTIDOMAIN supports MPI-based parallelism for simulating large problems based on the domain decomposition capabilities of the host grid wrapped by `MultiDomainGrid`. The authors of the neuron propagation simulation successfully employed this support to parallelize their application using an overlapping Schwarz decomposition, which allowed them to reduce the required time for a single simulation run with their default parameters from $\approx$ 19.2h to $\approx$ 2.52h by running the problem on 10 processors in parallel. This corresponds to a speedup of 7.58, which demonstrates that our framework does not introduce any fundamental obstacles to the creation of parallel programs. At the same time, the parallelization was mostly automatic; it was achieved by simply instructing the grid to loadbalance across the available processors and by switching to one of the standard parallel solver backends of PDELAB. The library and our extensions then automatically took

care of communicating and applying the required corrections for the overlapping Schwarz scheme.

The authors found that it was beneficial to partition the domain only in x-direction; that way, any processor boundaries will always be orthogonal to the membrane as shown in Figure 8.5. This setup avoids situations were a processor boundary coincides with a membrane boundary, which would severely complicate the implementation of the ODE solver for the ion channels, potentially requiring manual MPI communication to transport input and output data for that solver across the processor boundary. Moreover, restricting the parallel partitioning to the x-direction also precludes possible numerical issues caused by the high grid anisotropy in y-direction due to the Debye layer and its fine grid resolution in that direction.

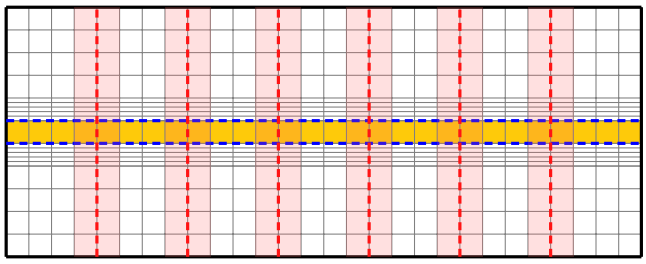


Figure 8.5 — Partition of the unmyelinated axon computational domain for the parallel case. The computational domain with its three subdomains is shown as before; additionally, processor boundaries (vertical dashed lines) and overlap elements (shaded) are shown, in this case exemplary for $p = 7$ processors. Note that this method gives optimal control over the load balancing and ensures that membrane interfaces never coincide with processor boundaries. – Image courtesy Jurgis Pods.

In conclusion, we can say that our software framework greatly contributed to the implementation of these simulations, reducing the time and effort required to arrive at a working simulation and allowing the authors to instead focus on model and method development. In particular, they were able to extend their model to describe the effects of myelination into their model: real axons are partly enclosed in a protective layer called myelin, and this layer has a significant impact on the extracellular electric field. Their extended model is capable of correctly modeling this change in behavior.

Finally, this example is interesting in that it stacks two meta grids on top of each other. In their work, the authors acknowledge the convenience of being able to extend the functionality of existing grids in this manner as well as the high level of integration with other parts of the PDELab and Dune-Multidomain stack (e.g. its seamless integration into the support for parallel simulations). At the same time, they note the substantial performance overhead imposed by the wrapping procedure, which they estimate at $\approx 30\%$ for each meta grid in the stack for their

application. This experience underlines the importance of improving the DUNE grid interface to reduce this performance impact.

CHAPTER 9

Conclusion

Developing Finite Element simulations for partial differential equations is a challenging task that requires a mixture of mathematics, computer science and application (model) knowledge, a unique combination of expertise that can not be expected from all researchers involved with PDEs. Scientific computing has alleviated some of the implementation burden by developing powerful solver toolboxes which provide high-level abstractions for crucial concepts like meshes, function spaces and integral forms and which interface with powerful linear algebra libraries for solving the resulting algebraic problems. However, existing frameworks focus almost exclusively on single domain problems.

Multi physics and in particular multi domain problems represent another step up the complexity ladder; they involve multiple domains (and thus meshes) as well as multiple variables and equations on subsets of those variables. In this work, we have created a flexible mathematical framework that describes the individual components of a multi domain problem and their interactions and presented a software implementation of these concepts (Chapter 4). The focus of this framework is on flexibility and the ability to experiment with different discretizations and solvers. To this end, we support a wide variety of Petrov-Galerkin methods (both continuous and discontinuous) that can be combined in arbitrary fashion.

At the same time, run time performance is of paramount importance for FEM simulations due to the sheer problem sizes. For this reason (and in keeping with the underlying DUNE framework), our implementation relies heavily on C++ templates, static polymorphism and function inlining. Unfortunately, infrastructure code based on these concepts tends to become very unwieldy and hard to maintain. For that reason, Chapter 5 has introduced a new template library called TypeTree for working with trees of heterogeneous objects. This library forms the basis of the composite function space implementations of both PDELab and our multi domain

extensions; its support for generic *tree transformations* is used to automatically derive a number of dependent object trees like local function spaces. Our extensions heavily exploit the modularity of those transformations to extend the functionality of standard PDELab without having to patch its source code by e.g. introducing additional types of function space tree nodes that are then automatically supported by the default tree transformations.

Before we can define function spaces on multiple domains, we need meshes that discretize those domains as well as a way to relate these meshes to each other. For this purpose, Chapter 3 has introduced a mechanism that extends existing grid managers with support for multiple subdomains inside an existing mesh. This approach makes it possible to use our framework on top of all grid managers supported by the DUNE framework and to choose the underlying grid based on application needs (structured vs. unstructured, simplicial vs. hexahedral, sequential vs. parallel etc.). Using a single "master grid" and carving it up into subdomains greatly simplifies rapid application development as it allows us to transfer data between subdomains via the master grid and thus avoids the highly complex problem of inter-grid data transfer between unrelated meshes.

While our work mostly concentrates on the problem *assembly*, i.e. the calculation of residual vectors and Jacobian matrices, we have also developed a versatile concept for controlling the structure of those vectors and matrices by means of the DOF ordering library described in Chapter 6. There is a wide variety of solution approaches for multi domain problems, which typically depend on a particular enumeration of the DOFs and / or a special block structure of the vectors and matrices. Users can create almost arbitrary DOF orders and block structures by simply annotating the nodes of their function space trees. Our implementation again relies on the TypeTree library to capture as much information as possible at compile time, allowing the compiler to generate highly optimized code. Due to the very performance-critical nature of the DOF mapping process during problem assembly, we have also presented and implemented a number of optimizations and caching mechanisms.

The ultimate goal of this work was to speed up the development of multi domain simulations by simplifying problem setup and, importantly, leveraging existing implementation code for single physics components of the overall problem. In Chapter 7, we have described a number of techniques that facilitate the reuse of existing subproblem-specific integration kernels without requiring those kernels to be aware of the larger, multi domain problem structure. This essentially involves rearranging and filtering the simulation data to fit the problem structure expected by the existing code. Importantly, we have shown how to achieve this data treatment with minimal run time and memory overhead.

Finally, in Chapter 8 we have demonstrated the feasibility of our approach by applying our framework to a number of model problems, investigating both the implementation effort (and possible code reuse) and the performance overhead

imposed by our implementation. There is a performance overhead, but is has proven to be within the bounds that are to be expected due to the additional work required to assemble a multi domain vs. a basic PDE problem. We have shown that the implementation also scales beyond those model problems by highlighting a number of external research results that have relied on our software to obtain novel results both at the methodical (mathematical) and the modeling (application) level.

In conclusion, our thesis and the associated software libraries greatly simplify the technical aspects of simulation development for multi domain problems, which makes it possible for application scientists to focus on their core interests of model and method development and to increase their productivity. The importance of providing this type of infrastructure cannot be underestimated, as evidenced by the substantial number of collaborations and users our work has already attracted during its development phase.

As mentioned before, our work has mostly focused on the assembly stage of PDE simulations; one straightforward extension of our framework would be improved support for the solution stage by providing a number of pre-packaged components for solvers that take advantage of the problem structure, e.g. a Dirichlet-Neumann iteration like the manually implemented example shown in Chapter 8, better support for problems with block structures of heterogeneous size or a framework for block preconditioners that still solves the fully coupled problem with a monolithic solver, but applies a different, problem-specific preconditioner to each subproblem.

We also limited ourselves to rudimentary support for parallel simulations. The individual subproblems in a multi domain simulation will in general not be balanced with regard to their computational complexity; one very important step towards efficient parallel multi domain simulations would be support for attaching appropriate weights to each subproblem and take those weights in account during loadbalancing. This will however require coordination within the DUNE project, as loadbalancing is handled by the grid manager; weighted loadbalancing would thus require changes to the canonical DUNE grid API.

Another issue that requires support from the DUNE community is the performance of `MultiDomainGrid`; as outlined in Chapter 3, a number of design decisions in the grid interface impose a significant performance overhead in relation to the host grid. In this area, we have developed a solution together with other core DUNE developers. As it requires a number of incompatible interface changes, it will be introduced with DUNE 3.0, which will be released in 2015.

Finally, it would be beneficial to further integrate Dune-Multidomain with the two other grid-level solutions Dune-Gridglue [20, 47] and Dune-UDG [46, 67] for multi domain problems in DUNE.

Code Examples

A.1 Strongly coupled Dirichlet-Neumann Operator for Poisson Problems

Listing A.1 — Coupling operator for strongly coupled Poisson problems

```
template<typename TReal>
class ContinuousValueContinuousFlowCoupling
  : public Dune::PDELab::MultiDomain::CouplingOperatorDefaultFlags
  , public Dune::PDELab::MultiDomain::NumericalJacobianCoupling<
      ContinuousValueContinuousFlowCoupling<TReal>
      >
  , public Dune::PDELab::MultiDomain::NumericalJacobianApplyCoupling<
      ContinuousValueContinuousFlowCoupling<TReal>
      >
  , public Dune::PDELab::MultiDomain::FullCouplingPattern
  , public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<TReal>
{

public:

  ContinuousValueContinuousFlowCoupling(
    int intorder,
    double intensity = 1.0)
    : _intorder(intorder)
    , _intensity(intensity)
  {}

  static const bool doAlphaCoupling = true;
  static const bool doPatternCoupling = true;

```

```
26  template<typename IG,
27           typename LFSU1, typename LFSU2,
28           typename LFSV1, typename LFSV2,
29           typename X, typename R>
30  void alpha_coupling(
31    const IG& ig,
32    const LFSU1& lfsu1, const X& x1, const LFSV1& lfsv1,
33    const LFSU2& lfsu2, const X& x2, const LFSV2& lfsv2,
34    R& r1, R& r2) const
35  {
36    // domain and range field type
37    typedef typename LFSU1::Traits::FiniteElementType::
38      Traits::LocalBasisType::Traits::DomainFieldType DF;
39    typedef typename LFSU1::Traits::FiniteElementType::
40      Traits::LocalBasisType::Traits::RangeFieldType RF;
41    typedef typename LFSU1::Traits::FiniteElementType::
42      Traits::LocalBasisType::Traits::RangeType RangeType;
43    typedef typename LFSU1::Traits::FiniteElementType::
44      Traits::LocalBasisType::Traits::JacobianType JacobianType;
45
46    typedef typename LFSU1::Traits::SizeType size_type;
47    typedef typename IG::Element Element;
48    typedef typename Element::Geometry::Jacobian GeometryJacobian;
49
50    const double h_F =
51      (ig.geometry().corner(0) - ig.geometry().corner(1)).two_norm();
52
53    // dimensions
54    const int dim = IG::dimension;
55
56    // select quadrature rule
57    Dune::GeometryType gtface = ig.geometryInInside().type();
58    const Dune::QuadratureRule<DF,dim-1>& rule =
59      Dune::QuadratureRules<DF,dim-1>::rule(gtface,_intorder);
60
61    // save entity pointers to adjacent elements
62    Element e1 = ig.insideElement();
63    Element e2 = ig.outsideElement();
64
65    // loop over quadrature points and integrate normal flux
66    for (auto it=rule.begin(); it!=rule.end(); ++it)
67      {
68        // position of quadrature point in local coordinates of element
69        Dune::FieldVector<DF,dim> local1 = ig.geometryInInside().global(it->position());
70        Dune::FieldVector<DF,dim> local2 = ig.geometryInOutside().global(it->position());
71
72        // evaluate ansatz shape functions (assume Galerkin for now)
73        std::vector<RangeType> phi1(lfsv1.size());
74        lfsv1.finiteElement().localBasis().evaluateFunction(local1,phi1);
75
76        std::vector<RangeType> phi2(lfsv2.size());
77        lfsv2.finiteElement().localBasis().evaluateFunction(local2,phi2);
```

```
78
79          // evaluate gradient of shape functions
80          std::vector<JacobianType> js1(lfsu1.size());
81          lfsu1.finiteElement().localBasis().evaluateJacobian(local1,js1);
82          std::vector<JacobianType> js2(lfsu2.size());
83          lfsu2.finiteElement().localBasis().evaluateJacobian(local2,js2);
84
85          // transform gradient to real element
86          const GeometryJacobian& jac1 = e1.geometry().jacobianInverseTransposed(local1);
87          const GeometryJacobian& jac2 = e2.geometry().jacobianInverseTransposed(local2);
88
89          std::vector<Dune::FieldVector<RF,dim> > gradphi1(lfsu1.size());
90          for (size_t i=0; i<lfsu1.size(); i++)
91            {
92              gradphi1[i] = 0.0;
93              jac1.umv(js1[i][0],gradphi1[i]);
94            }
95          std::vector<Dune::FieldVector<RF,dim> > gradphi2(lfsu2.size());
96          for (size_t i=0; i<lfsu2.size(); i++)
97            {
98              gradphi2[i] = 0.0;
99              jac2.umv(js2[i][0],gradphi2[i]);
100           }
101
102         // compute gradient of u1
103         Dune::FieldVector<RF,dim> gradu1(0.0);
104         for (size_t i=0; i<lfsu1.size(); i++)
105           gradu1.axpy(x1(lfsu1,i),gradphi1[i]);
106         // compute gradient of u2
107         Dune::FieldVector<RF,dim> gradu2(0.0);
108         for (size_t i=0; i<lfsu2.size(); i++)
109           gradu2.axpy(x2(lfsu2,i),gradphi2[i]);
110
111
112         RF u1(0.0);
113         for (size_t i=0; i<lfsu1.size(); i++)
114           u1 += x1(lfsu1,i) * phi1[i];
115
116         RF u2(0.0);
117         for (size_t i=0; i<lfsu2.size(); i++)
118           u2 += x2(lfsu2,i) * phi2[i];
119
120         Dune::FieldVector<DF,dim> normal1 = ig.unitOuterNormal(it->position());
121         Dune::FieldVector<DF,dim> normal2 = ig.unitOuterNormal(it->position());
122         normal2 *= -1;
123         const RF jump_u1 = u1 - u2;
124         const RF jump_u2 = u2 - u1;
125         Dune::FieldVector<RF,dim> mean_gradu = gradu1 + gradu2;
126         mean_gradu *= 0.5;
127         const double theta = 1;
128         const double alpha = _intensity;
129
```

```
        // integrate
        const RF factor = it->weight()*ig.geometry().integrationElement(it->position());
        for (size_type i=0; i<lfsv1.size(); i++)
          {
            RF value = 0.0;
            value -= /* epsilon * */ (mean_gradu * normal1) * phi1[i];
            value -= theta * /* epsilon * */ 0.5 * (gradphi1[i] * normal1) * jump_u1;
            value += alpha / h_F * jump_u1 * phi1[i];
            r1.accumulate(lfsv1,i,factor * value);
          }
        for (size_type i=0; i<lfsv2.size(); i++)
          {
            RF value = 0.0;
            value -= /* epsilon * */ (mean_gradu * normal2) * phi2[i];
            value -= theta * /* epsilon * */ 0.5 * (gradphi2[i] * normal2) * jump_u2;
            value += alpha / h_F * jump_u2 * phi2[i];
            r2.accumulate(lfsv2,i,factor * value);
          }
      }
  }

private:
  const int _intorder;
  const double _intensity;

};
```

A.2 Stokes-Darcy Coupling Operator

Listing A.2 — Coupling operator for Stokes-Darcy problem

```
template<typename Parameters>
class StokesDarcyCouplingOperator
  : public MultiDomain::CouplingOperatorDefaultFlags
  , public MultiDomain::FullCouplingPattern
  , public MultiDomain::NumericalJacobianCoupling<
      StokesDarcyCouplingOperator<Parameters>
      >
  , public Dune::PDELab::MultiDomain::NumericalJacobianApplyCoupling<
      StokesDarcyCouplingOperator<Parameters>
      >
{

public:

  static const bool doPatternCoupling = true;
  static const bool doAlphaCoupling = true;

  StokesDarcyCouplingOperator(const Parameters& params)
    : MultiDomain::NumericalJacobianCoupling<
        StokesDarcyCouplingOperator<Parameters>
```

```
21        >(params.epsilon())
22    , MultiDomain::NumericalJacobianApplyCoupling<
23        StokesDarcyCouplingOperator<Parameters>
24        >(params.epsilon())
25    , parameters(params)
26  {}
27
28  template<typename IG,
29           typename StokesLFSU,typename StokesLFSV,
30           typename DarcyLFSU, typename DarcyLFSV,
31           typename X, typename R>
32  void alpha_coupling(
33    const IG& ig,
34    const StokesLFSU& stokeslfsu, const X& stokesx, const StokesLFSV& stokeslsfv,
35    const DarcyLFSU& darcylfsu, const X& darcyx, const DarcyLFSV& darcylfsv,
36    R& stokesr, R& darcyr
37    ) const
38  {
39    // dimensions
40    const int dim = IG::dimension;
41    const int dimw = IG::dimensionworld;
42
43    // extract local function spaces
44    typedef typename StokesLFSU::template Child<0>::Type LFSU_V_PFS;
45    const LFSU_V_PFS& lfsu_v_pfs = stokeslfsu.template child<0>();
46
47    typedef typename LFSU_V_PFS::template Child<0>::Type LFSU_V;
48    const unsigned int vsize = lfsu_v_pfs.child(0).size();
49
50    // domain and range field type
51    typedef typename LFSU_V::Traits::FiniteElementType::
52      Traits::LocalBasisType::Traits::RangeFieldType RF;
53    typedef typename LFSU_V::Traits::FiniteElementType::
54      Traits::LocalBasisType::Traits::RangeType RT_V;
55    typedef typename LFSU_V::Traits::SizeType size_type;
56
57    typedef typename StokesLFSU::template Child<1>::Type LFSU_P;
58
59    typedef typename LFSU_P::Traits::FiniteElementType::
60      Traits::LocalBasisType::Traits::DomainFieldType DF;
61    typedef typename LFSU_P::Traits::FiniteElementType::
62      Traits::LocalBasisType::Traits::RangeType RT_P;
63
64    typedef typename DarcyLFSU::Traits::FiniteElementType::
65      Traits::LocalBasisType::Traits::RangeType RT_D;
66    typedef typename DarcyLFSU::Traits::FiniteElementType::
67      Traits::LocalBasisType::Traits::JacobianType JacobianType_D;
68    const unsigned int dsize = darcylfsu.size();
69
70    typedef typename IG::Geometry::LocalCoordinate LC;
71    typedef typename IG::Geometry::GlobalCoordinate GC;
72
```

```
73      // select quadrature rule
74      Dune::GeometryType gt = ig.geometry().type();
75      const int qorder = 2 * std::max(
76        lfsu_v_pfs.template child(0).finiteElement().localBasis().order(),
77        darcylfsu.finiteElement().localBasis().order()
78        );
79
80      const Dune::QuadratureRule<DF,dim-1>& rule =
81        Dune::QuadratureRules<DF,dim-1>::rule(gt,qorder);
82
83      const typename IG::Element& darcyCell = ig.outsideElement();
84
85      const RF g = parameters.gravity();
86      const RF alpha = parameters.alpha();
87      const RF nu = parameters.viscosity();
88      const RF porosity = parameters.porosity();
89      const RF gamma = parameters.gamma();
90      const RF rho = parameters.density();
91      const typename Parameters::PermeabilityTensor kabs = parameters.kabs();
92      RF tracePi = 0.0;
93      for (int i = 0; i < dim; ++i)
94        tracePi += kabs[i][i];
95      tracePi *= nu/g/rho;
96
97      // loop over quadrature points
98      for (typename Dune::QuadratureRule<DF,dim-1>::const_iterator it=rule.begin();
99           it!=rule.end();
100          ++it)
101       {
102
103         const GC pos = ig.geometry().global(it->position());
104         const GC stokesPos = ig.geometryInInside().global(it->position());
105         const GC darcyPos = ig.geometryInOutside().global(it->position());
106
107         // integration weight
108         const RF factor = it->weight() * ig.geometry().integrationElement(it->position());
109
110         std::vector<RT_V> v(vsize);
111         lfsu_v_pfs.child(0).finiteElement().localBasis().evaluateFunction(stokesPos,v);
112
113         std::vector<RT_D> psi(dsize);
114         darcylfsu.finiteElement().localBasis().evaluateFunction(darcyPos,psi);
115
116         // evaluate gradient of shape functions (we assume Galerkin method lfsu=lfsv)
117         std::vector<JacobianType_D> js(dsize);
118         darcylfsu.finiteElement().localBasis().evaluateJacobian(darcyPos,js);
119
120         // transform gradient to real element
121         const Dune::FieldMatrix<DF,dimw,dim> jac =
122           darcyCell.geometry().jacobianInverseTransposed(darcyPos);
123         std::vector<Dune::FieldVector<RF,dim> > gradpsi(dsize);
124         for (size_type i=0; i<vsize; i++)
```

```
125             jac.mv(js[i][0],gradpsi[i]);
126
127           // calculate phi and grad phi
128           RT_D phi = 0.0;
129           GC gradphi(0.0);
130           for (size_type i = 0; i < darcylfsu.size(); ++i)
131             {
132               phi += darcyx(darcylfsu,i) * psi[i];
133               gradphi.axpy(darcyx(darcylfsu,i),gradpsi[i]);
134             }
135
136           Dune::FieldVector<RF,dim> u(0.0);
137           const GC n = ig.unitOuterNormal(it->position());
138
139           // calculate u
140           for (int d = 0; d < dim; ++d)
141             {
142               const LFSU_V& lfsu_v = lfsu_v_pfs.child(d);
143               // calculate d-th component of u
144               for (size_type i = 0; i < lfsu_v.size(); ++i)
145                 u[d] += stokesx(lfsu_v,i) * v[i];
146             }
147
148           for (size_type i = 0; i < darcylfsu.size(); ++i)
149             darcyr.accumulate(darcylfsv,i, -gamma * porosity * (u * n) * psi[i] * factor);
150
151           Dune::FieldVector<RF,dim> tangentialFlow(0.0);
152           kabs.mv(gradphi,tangentialFlow);
153           tangentialFlow /= porosity;
154           tangentialFlow += u;
155           // project into tangential plane
156           GC scaledNormal = n;
157           scaledNormal *= (tangentialFlow * n);
158           tangentialFlow -= scaledNormal;
159
160           for (int d = 0; d < dim; ++d)
161             {
162               const LFSU_V& lfsu_v = lfsu_v_pfs.child(d);
163               for (size_type i = 0; i < lfsu_v.size(); ++i)
164                 {
165                   stokesr.accumulate(
166                     lfsu_v,
167                     i,
168                     - rho * g * (phi - pos[dim-1]) * v[i] * n[d] * factor);
169                 }
170
171               for (size_type i = 0; i < lfsu_v.size(); ++i)
172                 {
173                   stokesr.accumulate(
174                     lfsu_v,
175                     i,
176                     alpha * sqrt(dim) / sqrt(tracePi) * tangentialFlow[d] * v[i] * factor);
```

```
177                 }
178               }
179             }
180   }
181
182 private:
183
184   const Parameters& parameters;
185 };
```

A.3 Neumann-Dirichlet Coupling Operator for Poisson Problems

Listing A.3 — Neumann-Dirichlet coupling operator for Poisson oroblems

```
1  template<typename T>
2  class ConvectionDiffusionDGNeumannDirichletCoupling
3    : public Dune::PDELab::MultiDomain::CouplingOperatorDefaultFlags
4    , public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<
5        typename T::Traits::RangeFieldType
6        >
7  {
8    enum { dim = T::Traits::GridViewType::dimension };
9
10   typedef typename T::Traits::RangeFieldType Real;
11   typedef typename ConvectionDiffusionBoundaryConditions::Type BCType;
12
13 public:
14   // pattern assembly flags
15   // there is no additional pattern relative to the subproblems
16
17   // residual assembly flags
18   enum { doAlphaCoupling  = true };
19
20   //! constructor: pass parameter object
21   ConvectionDiffusionDGNeumannDirichletCoupling(
22     CouplingMode coupling_mode,
23     T& param_,
24     ConvectionDiffusionDGMethod::Type method_,
25     Real alpha_=0.0,
26     int intorderadd_=0)
27     : _coupling_mode(coupling_mode)
28     , param(param_)
29     , method(method_)
30     , alpha(alpha_)
31     , intorderadd(intorderadd_)
32     , quadrature_factor(2)
33   {
34     theta = 1.0;
35     if (method==ConvectionDiffusionDGMethod::SIPG) theta = -1.0;
```

```
36    }
37
38    // coupling integral depending on test and ansatz functions
39    template<
40      typename IG,
41      typename LFSU, typename LFSV,
42      typename RemoteLFSU, typename RemoteLFSV,
43      typename X, typename R
44      >
45    void alpha_coupling(
46      const IG& ig,
47      const LFSU& lfsu_s, const X& x_s, const LFSV& lfsv_s,
48      const RemoteLFSU& lfsu_r, const X& x_r, const RemoteLFSV& lfsv_r,
49      R& r_s, const R& r_r) const
50    {
51      // domain and range field type
52      typedef typename LFSV::Traits::FiniteElementType::
53        Traits::LocalBasisType::Traits::DomainFieldType DF;
54      typedef typename LFSV::Traits::FiniteElementType::
55        Traits::LocalBasisType::Traits::RangeFieldType RF;
56      typedef typename LFSV::Traits::FiniteElementType::
57        Traits::LocalBasisType::Traits::RangeType RangeType;
58      typedef typename LFSU::Traits::FiniteElementType::
59        Traits::LocalBasisType::Traits::JacobianType JacobianType;
60      typedef typename LFSV::Traits::SizeType size_type;
61
62      // dimensions
63      const int dim = IG::dimension;
64      const int order = std::max(
65          std::max(lfsu_s.finiteElement().localBasis().order(),
66              lfsu_r.finiteElement().localBasis().order()),
67          std::max(lfsv_s.finiteElement().localBasis().order(),
68              lfsv_r.finiteElement().localBasis().order())
69          );
70      const int intorder = intorderadd+quadrature_factor*order;
71
72      // evaluate permeability tensors
73      auto inside_local = Dune::ReferenceElements<DF,dim>::general(
74        ig.inside()->type()
75        ).position(0,0);
76      auto outside_local = Dune::ReferenceElements<DF,dim>::general(
77        ig.outside()->type()
78        ).position(0,0);
79      typename T::Traits::PermTensorType A_s, A_n;
80      A_s = param.A(*(ig.inside()),inside_local);
81      A_n = param.A(*(ig.outside()),outside_local);
82
83      // face diameter
84      DF h_s, h_n;
85      DF hmax_s = 0.;
86      DF hmax_n = 0.;
87      element_size(ig.inside()->geometry(),h_s,hmax_s);
```

```
88      element_size(ig.outside()->geometry(),h_n,hmax_n);
89      RF h_F = std::min(h_s,h_n);
90      h_F = std::min(
91        ig.inside()->geometry().volume(),
92        ig.outside()->geometry().volume()
93        ) / ig.geometry().volume();
94
95      // select quadrature rule
96      auto gtface = ig.geometryInInside().type();
97      const Dune::QuadratureRule<DF,dim-1>& rule =
98        Dune::QuadratureRules<DF,dim-1>::rule(gtface,intorder);
99
100     // transformation
101     typename IG::Entity::Geometry::JacobianInverseTransposed jac;
102
103     // tensor times normal
104     const Dune::FieldVector<DF,dim> n_F = ig.centerUnitOuterNormal();
105     Dune::FieldVector<RF,dim> An_F_s;
106     A_s.mv(n_F,An_F_s);
107     Dune::FieldVector<RF,dim> An_F_n;
108     A_n.mv(n_F,An_F_n);
109
110     // weights
111     RF omega_s = 0.5;
112     RF omega_n = 0.5;
113     RF harmonic_average = 1.0
114
115     // get polynomial degree
116     const int order_s = lfsu_s.finiteElement().localBasis().order();
117     const int order_n = lfsu_r.finiteElement().localBasis().order();
118     int degree = std::max( order_s, order_n );
119
120     // penalty factor
121     RF penalty_factor = (alpha/h_F) * harmonic_average * degree*(degree+dim-1);
122
123     // loop over quadrature points
124     for (auto it=rule.begin(); it!=rule.end(); ++it)
125       {
126         // exact normal
127         const auto n_F_local = ig.unitOuterNormal(it->position());
128
129         // position of quadrature point in local coordinates of elements
130         auto iplocal_s = ig.geometryInInside().global(it->position());
131         auto iplocal_n = ig.geometryInOutside().global(it->position());
132
133
134         // evaluate basis functions
135         std::vector<RangeType> psi_s(lfsv_s.size());
136         lfsv_s.finiteElement().localBasis().evaluateFunction(iplocal_s,psi_s);
137
138         // integration factor
139         RF factor = it->weight() * ig.geometry().integrationElement(it->position());
```

```
140
141         // evaluate velocity field
142         auto b = param.b(*(ig.inside()),iplocal_s);
143         RF normalflux = b*n_F_local;
144
145         // evaluate basis functions
146         std::vector<RangeType> phi_r(lfsu_r.size());
147         lfsu_r.finiteElement().localBasis().evaluateFunction(iplocal_n,phi_r);
148
149         // calculate solution in remote domain
150         RF u_r=0.0;
151         for (size_type i=0; i<lfsu_r.size(); i++)
152           u_r += x_r(lfsu_r,i)*phi_r[i];
153
154         if (_coupling_mode == CouplingMode::Neumann)
155           {
156             // evaluate flux in remote domain
157             std::vector<JacobianType> gradphi_r(lfsu_r.size());
158             lfsu_r.finiteElement().localBasis().evaluateJacobian(iplocal_n,gradphi_r);
159
160             jac = ig.outside()->geometry().jacobianInverseTransposed(iplocal_n);
161             std::vector<Dune::FieldVector<RF,dim> > tgradphi_r(lfsu_r.size());
162             for (size_type i=0; i<lfsu_r.size(); i++)
163               jac.mv(gradphi_r[i][0],tgradphi_r[i]);
164
165             Dune::FieldVector<RF,dim> gradu_r(0.0);
166             for (size_type i=0; i<lfsu_r.size(); i++)
167               gradu_r.axpy(x_r(lfsu_r,i),tgradphi_r[i]);
168
169             // flux in normal direction
170             RF j = normalflux * u_r - gradu_r * n_F_local;
171
172             // integrate
173             for (size_type i=0; i<lfsv_s.size(); i++)
174               r_s.accumulate(lfsv_s,i, j * psi_s[i] * factor);
175
176             // jump to next quadrature point
177             continue;
178           }
179
180         std::vector<RangeType> phi_s(lfsu_s.size());
181         lfsu_s.finiteElement().localBasis().evaluateFunction(iplocal_s,phi_s);
182
183         // evaluate u in local subdomain
184         RF u_s=0.0;
185         for (size_type i=0; i<lfsu_s.size(); i++)
186           u_s += x_s(lfsu_s,i)*phi_s[i];
187
188         // evaluate gradient of basis functions (we assume Galerkin method lfsu=lfsv)
189         std::vector<JacobianType> gradphi_s(lfsu_s.size());
190         lfsu_s.finiteElement().localBasis().evaluateJacobian(iplocal_s,gradphi_s);
191         std::vector<JacobianType> gradpsi_s(lfsv_s.size());
```

```
192         lfsv_s.finiteElement().localBasis().evaluateJacobian(iplocal_s,gradpsi_s);
193
194         // transform gradients of shape functions to real element
195         jac = ig.inside()->geometry().jacobianInverseTransposed(iplocal_s);
196         std::vector<Dune::FieldVector<RF,dim> > tgradphi_s(lfsu_s.size());
197         for (size_type i=0; i<lfsu_s.size(); i++)
198           jac.mv(gradphi_s[i][0],tgradphi_s[i]);
199         std::vector<Dune::FieldVector<RF,dim> > tgradpsi_s(lfsv_s.size());
200         for (size_type i=0; i<lfsv_s.size(); i++)
201           jac.mv(gradpsi_s[i][0],tgradpsi_s[i]);
202
203         // compute gradient of u
204         Dune::FieldVector<RF,dim> gradu_s(0.0);
205         for (size_type i=0; i<lfsu_s.size(); i++)
206           gradu_s.axpy(x_s(lfsu_s,i),tgradphi_s[i]);
207
208         // upwinding
209         RF omegaup_s = normalflux >= 0.0 ? 1.0 : 0.0;
210         RF omegaup_r = normalflux >= 0.0 ? 0.0 : 1.0;
211
212         // convection term
213         RF term1 = (omegaup_s*u_s + omegaup_r*u_r) * normalflux *factor;
214         for (size_type i=0; i<lfsv_s.size(); i++)
215           r_s.accumulate(lfsu_s,i,term1 * psi_s[i]);
216
217         // diffusion term
218         RF term2 =  -factor * (An_F_s*gradu_s);
219         for (size_type i=0; i<lfsv_s.size(); i++)
220           r_s.accumulate(lfsv_s,i,term2 * psi_s[i]);
221
222         // (non-)symmetric IP term
223         RF term3 = (u_s-u_r) * factor;
224         for (size_type i=0; i<lfsv_s.size(); i++)
225           r_s.accumulate(lfsv_s,i,term3 * theta * (An_F_s*tgradpsi_s[i]));
226
227         // standard IP term integral
228         RF term4 = penalty_factor * (u_s-u_r) * factor;
229         for (size_type i=0; i<lfsv_s.size(); i++)
230           r_s.accumulate(lfsv_s,i,term4 * psi_s[i]);
231       }
232   }
233
234 private:
235   // member variables omitted for brevity
236
237   template<typename GEO, typename ctype>
238   void element_size (const GEO& geo, ctype& hmin, ctype hmax) const
239   {
240     // calculate element size
241   }
242 };
```

Hardware Configurations

B.1 Configuration A

Name	Details
CPU	Intel Core 2 Duo Dual Core (T9600) 2.8 GHz 8 GiB RAM
GPU	NVIDIA GeForce 9600M GT 512 MiB RAM
Operating system	Mac OS X 10.7 Lion

B.2 Configuration B

Name	Details
CPU	Intel Core i7 Quad Core (4960HQ) 2.6 GHz 16 GiB RAM
GPU	NVIDIA GeForce 750 M 2 GiB RAM
Operating system	Mac OS X 10.9 Mavericks

Bibliography

[1] G. Aigner and U. Hölzle. "Eliminating virtual function calls in C++ programs". In: *ECOOP '96 — Object-Oriented Programming*. Ed. by P. Cointe. Vol. 1098. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 142–166. DOI: 10.1007/BFb0053060. — [70]

[2] *ALBERTA web site*. URL: http://www.alberta-fem.de (visited on 06/28/2013). — [4]

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. 1st ed. C++ In-Depth Series. Addison-Wesley Professional, 2001. — [31, 71]

[4] *ALUGrid web site*. URL: http://aam.mathematik.uni-freiburg.de/IAM/Research/alugrid/ (visited on 06/28/2013). — [4]

[5] J.-P. Aumasson and D. J. Bernstein. "SipHash: A Fast Short-Input PRF". In: *Progress in Cryptology – INDOCRYPT 2012*. Ed. by S. Galbraith and M. Nandi. Vol. 7668. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 489–508. DOI: 10.1007/978-3-642-34931-7_28. — [115]

[6] J.-P. Aumasson and D. J. Bernstein. *SipHash Website*. URL: https://131002.net/siphash/ (visited on 06/16/2013). — [115]

[7] K. Baber. "Coupling free flow and flow in porous media in biological and technical applications: From a simple to a complex interface description". PhD thesis. Universität Stuttgart, 2014. — [129]

[8] K. Baber, K. Mosthaf, B. Flemisch, R. Helmig, S. Müthing, and B. Wohlmuth. "Numerical scheme for coupling two-phase compositional porous-media flow and one-phase compositional free flow". In: *IMA Journal of Applied Mathematics* 77.6 (2012), pp. 887–909. DOI: 10.1093/imamat/hxs048. eprint: http://imamat.oxfordjournals.org/content/77/6/887.full.pdf+html. — [129]

[9] L. Badea, M. Discacciati, and A. Quarteroni. "Numerical analysis of the Navier–Stokes/Darcy coupling". In: *Numerische Mathematik* 115 (2 2010). 10.1007/s00211-009-0279-6, pp. 195–227. URL: http://dx.doi.org/10.1007/s00211-009-0279-6. — [137]

[10] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.2. Argonne National Laboratory, 2011. — [5, 18]

[11] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. *PETSc Web page*. 2011. URL: http://www.mcs.anl.gov/petsc (visited on 05/26/2013). — [5, 18]

[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries”. In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202. — [5, 18]

[13] W. Bangerth, R. Hartmann, and G. Kanschat. *deal.II Differential Equations Analysis Library, Technical Reference*. http://www.dealii.org. — [4]

[14] 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”. In: *Computing and Visualization in Science* 1.1 (1997), pp. 27–40. DOI: 10.1007/s007910050003. — [4]

[15] P. Bastian, F. Heimann, and S. Marnach. “Generic implementation of finite element methods in the Distributed and Unified Numerics Environment (DUNE)”. In: *Kybernetika* 46.2 (2010), pp. 294–315. — [22, 104, 114]

[16] P. Bastian. *Lecture Notes on Scientific Computing with Partial Differential Equations*. 2014. URL: http://conan.iwr.uni-heidelberg.de/teaching/numerik2_ss2014/num2.pdf (visited on 08/23/2014). — [7, 10, 16]

[17] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, C. Wieners, G. Wittum, and C. Wrobel. “A Parallel Software-Platform for Solving Problems of Partial Differential Equations using Unstructured Grids and Adaptive Multigrid Methods”. In: *High Performance Computing in Science and Engineering ’98*. Ed. by E. Krause and W. Jäger. Springer Berlin Heidelberg, 1999, pp. 326–339. DOI: 10.1007/978-3-642-58600-2_31. — [4]

[18] 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 I: Abstract Framework”. In: *Computing* 82.2-3 (2008), pp. 103–119. — [20, 23, 101]

[19] 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”. In: *Computing* 82.2-3 (2008), pp. 121–138. — [20, 23, 101]

[20] P. Bastian, G. Buse, and O. Sander. “Infrastructure for the Coupling of Dune Grids”. In: *Proceedings of ENUMATH 2009*. 2010, pp. 107–114. — [35, 147]

[21] G. S. Beavers and D. D. Joseph. “Boundary conditions at a naturally permeable wall”. In: *J. Fluid Mech* 30.1 (1967), pp. 197–207. — [53]

[22] M. Blatt and P. Bastian. “On the generic parallelisation of iterative solvers for the finite element method”. In: *Int. J. Comput. Sci. Eng.* 4.1 (Nov. 2008), pp. 56–69. DOI: 10.1504/IJCSE.2008.021112. — [6]

[23] M. Blatt and P. Bastian. "The Iterative Solver Template Library". In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Ed. by B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski. Vol. 4699. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 666–675. DOI: 10.1007/978-3-540-75755-9_82. — [6]

[24] R. H. Brooks and A. T. Corey. "Hydraulic properties of porous media". In: *Hydrology Paper* 3 (1964). Fort Collins: Colorado State University, pp. 27–110. — [56]

[25] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. "An efficient implementation of an adaptive and parallel grid in DUNE". In: *Computational Science and High Performance Computing II*. Ed. by E. Krause, Y. Shokin, M. Resch, and N. Shokina. Notes on Numerical Fluid Mechanics and Multidisciplinary Design 91. Springer Berlin Heidelberg, 2006, pp. 67–82. DOI: 10.1007/3-540-31768-6_7. — [4]

[26] F. Büttner, O. Radfelder, A. Lindow, and M. Gogolla. "Digging into the visitor pattern". In: *Proc. of International Conference on Software Engineering & Knowledge Engineering (SEKE)*. Citeseer. 2004. — [80]

[27] *C++ B-tree Library*. URL: https://code.google.com/p/cpp-btree/ (visited on 11/10/2014). — [70]

[28] B. Calder and D. Grunwald. "Reducing indirect function call overhead in C++ programs". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1994, pp. 397–408. — [70]

[29] Y. Cao, M. Gunzburger, X. Hu, F. Hua, X. Wang, and W. Zhao. "Finite Element Approximations for Stokes-Darcy Flow with Beavers-Joseph Interface Conditions". In: *SIAM Journal on Numerical Analysis* 47.6 (2010), pp. 4239–4256. — [52, 136]

[30] P. Chidyagwai and B. Rivière. "Numerical modelling of coupled surface and subsurface flow systems". In: *Advances in Water Resources* 33.1 (2010), pp. 92–105. — [137]

[31] *CityHash website*. URL: http://code.google.com/p/cityhash/ (visited on 06/16/2013). — [115]

[32] *COMSOL Multiphysics*. COMSOL Inc. 2013. URL: http://www.comsol.com. — [104]

[33] E. Cuthill and J. McKee. "Reducing the bandwidth of sparse symmetric matrices". In: *Proceedings of the 1969 24th national conference*. ACM '69. ACM, 1969, pp. 157–172. DOI: 10.1145/800195.805928. — [111]

[34] 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*. Preprint No. 3. Mathematisches Institut, Universität Freiburg. 2009. — [22]

[35] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. "A supernodal approach to sparse partial pivoting". In: *SIAM Journal on Matrix Analysis and Applications* 20.3 (1999), pp. 720–755. — [111, 137]

[36] *Diffpack web site*. URL: http://www.diffpack.com (visited on 06/29/2013). — [4]

[37] M. Discacciati, E. Miglio, and A. Quarteroni. "Mathematical and numerical models for coupling surface and groundwater flows". In: *Applied Numerical Mathematics* 43.1-2 (2002), pp. 57–74. DOI: DOI:10.1016/S0168-9274(02)00125-3. — [137]

[38] M. Discacciati and A. Quarteroni. "Convergence analysis of a subdomain iterative method for the finite element approximation of the coupling of Stokes and Darcy equations". In: *Computing and Visualization in Science* 6 (2 2004). 10.1007/s00791-003-0113-0, pp. 93–103. URL: http://dx.doi.org/10.1007/s00791-003-0113-0. — [137]

[39] 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. San Jose, California, USA: ACM, 1996, pp. 306–323. DOI: 10.1145/236337.236369. — [70]

[40] *DUNE download and licence page*. URL: http://www.dune-project.org/download.html (visited on 05/31/2013). — [36, 50, 69]

[41] *DUNE-FEM web site*. URL: http://dune.mathematik.uni-freiburg.de (visited on 06/29/2013). — [22]

[42] *dune-foamgrid web site*. URL: http://users.dune-project.org/projects/dune-foamgrid (visited on 09/16/2014). — [22]

[43] *dune-spgrid web site*. URL: http://dune.mathematik.uni-freiburg.de/grids/dune-spgrid/ (visited on 09/16/2014). — [22]

[44] T. Dunne, R. Rannacher, and T. Richter. "Fundamental Trends in Fluid-Structure Interaction". In: Contemporary Challenges in Mathematical Fluid Dynamics and Its Applications 1. World Scientific Publishing, 2010. Chap. Numerical Simulation of Fluid-Structure Interaction Based on Monolithic Variational Formulations, pp. 1–77. — [2]

[45] H. Edwards. "Managing complexity in massively parallel, adaptive, multiphysics applications". In: *Engineering with Computers* 22 (3 2006), pp. 135–155. — [4, 35]

[46] C. Engwer and F. Heimann. "Dune-UDG: A Cut-Cell Framework for Unfitted Discontinuous Galerkin Methods". In: *Advances in DUNE*. Ed. by A. Dedner, B. Flemisch, and R. Klöfkorn. Springer Berlin Heidelberg, 2012, pp. 89–100. DOI: 10.1007/978-3-642-28589-9_7. — [147]

[47] C. Engwer and S. Müthing. "Concepts for flexible parallel multi-domain simulations". In: *Domain Decomposition Methods in Science and Engineering 22*. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, to appear. — [35, 147]

[48] A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*. Applied Mathematical Sciences 159. Springer-Verlag New York, 2004. — [15]

[49] A. Ern, A. F. Stephansen, and P. Zunino. "A discontinuous Galerkin method with weighted averages for advection—diffusion equations with locally small and anisotropic diffusivity". In: *IMA Journal of Numerical Analysis* 29.2 (2009), pp. 235–256. DOI: 10.1093/imanum/drm050. eprint: http://imajna.oxfordjournals.org/content/29/2/235.full.pdf+html. — [54, 136]

[50] B. Flemisch, M. Darcis, K. Erbertseder, B. Faigle, A. Lauser, K. Mosthaf, S. Müthing, P. Nuske, A. Tatomir, M. Wolff, et al. "DuMux: DUNE for Multi-{Phase, Component, Scale, Physics, ...} Flow and Transport in Porous Media". In: *Advances in Water Resources* 34.9 (2011), pp. 1102–1112. DOI: 10.1016/j.advwatres.2011.03.007. — [137]

[51] G. Fowler, L. C. Noll, and P. Vo. *FNV hash algorithm*. 1991. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html (visited on 06/16/2013). — [115]

[52] A. George, J. R. Gilbert, and J. W. Liu, eds. *Graph theory and sparse matrix computation*. Vol. 56. Springer Verlag, 1993. — [111]

[53] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981. — [111]

[54] V. Girault and B. Rivière. "DG Approximation of Coupled Navier-Stokes and Darcy Equations by Beaver-Joseph-Saffman Interface Condition". In: *SIAM J. Numer. Anal.* 47.3 (2009), pp. 2052–2089. DOI: DOI:10.1137/070686081. — [137]

[55] V. Girault, S. Shuyu, M. F. Wheeler, and I. Yotov. "Coupling Discontinuous Galerking and mixed finite element discretizations using mortar finite elements." In: *SIAM Journal on Numerical Analysis* 46.2 (2008), pp. 949–979. URL: http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx?direct=true&db=aph&AN=31380724&site=ehost-live. — [137]

[56] P. Gottschling, D. S. Wise, and M. D. Adams. "Representation-transparent matrix algorithms with scalable performance". In: *Proceedings of the 21st annual international conference on Supercomputing*. ICS '07. Seattle, Washington: ACM, 2007, pp. 116–125. DOI: 10.1145/1274971.1274989. — [5]

[57] C. Gräser, U. Sack, and O. Sander. *dune-fufem on DUNE web site*. URL: http://www.dune-project.org/discmodule.html (visited on 09/16/2014). — [22]

[58] C. Gräser and O. Sander. "The dune-subgrid module and Some Applications". In: *Computing* 8.4 (2009), pp. 269–290. — [42]

[59] D. Gregor and J. Järvi. "Variadic templates for C++". In: *Proceedings of the 2007 ACM symposium on Applied computing*. SAC '07. Seoul, Korea: ACM, 2007, pp. 1101–1108. DOI: 10.1145/1244002.1244243. — [32]

[60] G. Guennebaud, B. Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org. — [5]

[61] A. Gurtovoy and D. Abrahams. *The Boost MPL Library*. 2002–2004. URL: http://www.boost.org/libs/mpl/. — [33, 71]

[62] J. de Guzman, D. Marsden, and T. Schwinger. *The Boost Fusion Library*. 2001–2012. URL: http://www.boost.org/libs/fusion/. — [71, 77]

[63] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics 14. Springer Berlin Heidelberg, 2010. — [13]

[64] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Series in Computational Mathematics 8. Springer Berlin Heidelberg, 2011. — [13]

[65] F. Hecht. "C++ Tools to construct our user-level language". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 36.5 (Aug. 2002), pp. 809–836. DOI: 10.1051/m2an:2002034. — [4]

[66] F. Hecht. *Freefem++ manual*. 3rd ed. Version 3.22. June 2013. URL: http://www.freefem.org/ff++/ftp/freefem++doc.pdf. — [4]

[67] F. Heimann, C. Engwer, O. Ippisch, and P. Bastian. "An unfitted interior penalty discontinuous Galerkin method for incompressible Navier-Stokes two-phase flow". In: *International Journal for Numerical Methods in Fluids* 71.3 (2013), pp. 269–293. DOI: 10.1002/fld.3653. — [147]

[68] 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, et al. "An overview of the Trilinos project". In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 397–423. DOI: http://doi.acm.org/10.1145/1089014.1089021. — [5, 18]

[69] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of Physiology* 117.4 (1952), pp. 500–544. — [139]

[70] P. Hudak. "Conception, evolution, and application of functional programming languages". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. DOI: 10.1145/72551.72554. — [83]

[71] A. Inc. *Adobe Forest Tree Library*. URL: http://stlab.adobe.com/group__forest__related.html (visited on 11/10/2014). — [70]

[72] International Organization for Standardization. *ISO/IEC 14882:2003 Programming Language C++*. Oct. 2003. — [27, 71]

[73] International Organization for Standardization. *ISO/IEC 14882:2011 Programming Language C++*. Sept. 2011. — [27]

[74] W. Jäger and A. Mikelić. "Modeling Effective Interface Laws for Transport Phenomena Between an Unconfined Fluid and a Porous Medium Using Homogenization". In: *Transport in Porous Media* 78.3 (2009), pp. 489–508. — [53]

[75] I. P. Jones. "Low Reynolds number flow past a porous spherical shell". In: *Cambridge Philosophical Society, Proceedings*. Vol. 73. Cambridge Univ Press. 1973, pp. 231–238. — [53]

[76] G. Kanschat and B. Rivière. "A strongly conservative finite element method for the coupling of Stokes and Darcy flow". In: *Journal of Computational Physics* 229.17 (2010), pp. 5933–5943. — [137]

[77] V. Karvonen and P. Mensonides. *The Boost Preprocesser Library*. 2001. URL: http://www.boost.org/libs/preprocessor/. — [32]

[78] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. 2nd ed. Vol. 1. Redwood City, CA, USA: Addison-Wesley, 1997. — [70]

[79] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. 2nd ed. Vol. 3. Redwood City, CA, USA: Addison-Wesley, 1998. — [115]

[80] H. P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Lecture Notes in Computational Science and Engineering 2. Springer-Verlag New York, 2003. — [4]

[81] A. Logg. "Automating the Finite Element Method". In: *Arch. Comput. Methods Eng.* 14.2 (2007), pp. 93–138. — [104]

[82] A. Logg, K.-A. Mardal, and G. Wells. *Automated solution of Differential Equations by the Finite Element Method*. Springer, 2012. — [4, 15]

[83] K. Long, R. Kirby, and B. van Bloemen Waanders. "Unified Embedded Parallel Finite Element Computations via Software-Based Fréchet Differentiation". In: *SIAM Journal on Scientific Computing* 32.6 (2010), pp. 3323–3351. DOI: 10.1137/09076920X. eprint: http://epubs.siam.org/doi/pdf/10.1137/09076920X. — [4]

[84] B. Lu, M. J. Holst, J. A. McCammon, and Y. Zhou. "Poisson–Nernst–Planck equations for simulating biomolecular diffusion–reaction processes I: Finite element solutions". In: *Journal of Computational Physics* 229.19 (2010), pp. 6979–6994. DOI: 10.1016/j.jcp.2010.05.035. — [138]

[85] A. Lumsdaine, J. Siek, L.-Q. Lee, and P. Gottschling. *Matrix Template Library web site*. 2006. URL: http://osl.iu.edu/research/mtl/. — [5]

[86] Y. Mori. "From three-dimensional electrophysiology to the cable model: an asymptotic study". In: *arXiv preprint arXiv:0901.3914* (2009). — [138]

[87] K. Mosthaf, K. Baber, B. Flemischh, R. Helmig, A. Leijnse, I. Rybak, and B. Wohlmuth. "A coupling concept for two-phase compositional porous-medium and single-phase compositional free flow". In: *Water Resources Research* 47.10 (2011). DOI: 10.1029/2011WR010685. — [129, 137]

[88] K. Mosthaf. "Modeling and analysis of coupled porous-medium and free flow with application to evaporation processes". PhD thesis. Universität Stuttgart, 2014. — [129, 137]

[89] *MpCCI Website*. Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen SCAI. URL: http://www.mpcci.de (visited on 06/08/2013). — [4, 35]

[90] *MurmurHash Website*. URL: http://code.google.com/p/smhasher/wiki/MurmurHash3 (visited on 06/16/2013). — [115]

[91] S. Müthing. *dune-multidomain 2.0.1*. Dec. 2014. DOI: 10.5281/zenodo.13193. — [50]

[92] S. Müthing. *dune-multidomain web site*. URL: https://github.com/smuething/dune-multidomain (visited on 05/31/2013). — [50]

[93] S. Müthing. *dune-multidomaingrid 2.3.1*. Nov. 2014. DOI: 10.5281/zenodo.12887. — [36]

[94] S. Müthing. *dune-multidomaingrid web site*. URL: https://github.com/smuething/dune-multidomaingrid (visited on 09/16/2014). — [36]

[95] S. Müthing. *TypeTree web site*. URL: https://github.com/smuething/dune-typetree (visited on 10/18/2014). — [69]

[96] S. Müthing and P. Bastian. "Dune-Multidomaingrid: A Metagrid Approach to Subdomain Modeling". In: *Advances in DUNE*. Ed. by A. Dedner, B. Flemisch, and R. Klöfkorn. Springer Berlin Heidelberg, 2012, pp. 59–73. DOI: 10.1007/978-3-642-28589-9_5. — [36]

[97] S. Müthing, M. Blatt, D. Kempf, B. Skaflestad, A. Buhr, and A. Burchardt. *TypeTree 2.3.1*. Nov. 2014. DOI: 10.5281/zenodo.10304. — [69]

[98] M. Nolte. "Efficient Numerical Approximation of the Effective Hamiltonian". PhD thesis. Albert-Ludwigs-Universität Freiburg, 2011. — [22]

[99] J. Palsberg and C. Jay. "The essence of the Visitor pattern". In: *The Twenty-Second Annual International Computer Software and Applications Conference*. 1998, pp. 9–15. DOI: 10.1109/CMPSAC.1998.716629. — [80]

[100] S. Parter. "The Use of Linear Graphs in Gauss Elimination". In: *SIAM Review* 3.2 (1961), pp. 119–130. URL: http://www.jstor.org/stable/2027387. — [111]

[101] PDELab Team. *PDELab Version 2.0.0 Howto*. URL: http://www.dune-project.org/pdelab/pdelab-howto-2.0.0.pdf (visited on 10/17/2014). — [18, 24]

[102] *PDELab web site*. URL: http://www.dune-project.org/pdelab/ (visited on 09/16/2014). — [22, 24]

[103] K. Peeters. *tree.hh: an STL-like C++ tree class*. URL: http://tree.phi-sci.com (visited on 11/09/2014). — [70]

[104] J. Pods. "Electrodiffusion Models of Axon and Extracellular Space Using the Poisson-Nernst-Planck Equations". PhD thesis. Universität Heidelberg, 2014. — [137]

[105] J. Pods, J. Schönke, and P. Bastian. "Electrodiffusion Models of Neurons and Extracellular Space Using the Poisson-Nernst-Planck Equations – Numerical Simulation of the Intra- and Extracellular Potential for an Axon Model". In: *Biophysical Journal* 105.1 (2013), pp. 242–254. — [137]

[106] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav. "Compiler optimization of C++ virtual function calls". In: *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*. COOTS'96. Toronto, Ontario, Canada: USENIX Association, 1996, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1268049.1268050. — [70]

[107] *PreCICE web site*. URL: http://www5.in.tum.de/wiki/index.php/PreCICE_Webpage (visited on 11/14/2014). — [4]

[108] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, 1999. — [2]

[109] A. Rasmussen. *dune-cornerpoint web site*. URL: https://github.com/OPM/dune-cornerpoint (visited on 09/16/2014). — [22]

[110] B. Rivière and I. Yotov. "Locally Conservative Couplings of Stokes and Darcy Flows". In: *SIAM Journal on Numerical Analysis* 42.5 (2005), pp. 1959–1977. — [137]

[111] P. Saffman. "On the boundary condition at the interface of a porous medium". In: *Stud. Appl. Math.* 1 (1971), pp. 77–84. — [53]

[112] A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software*. Lecture Notes in Computational Science and Engineering 42. Springer, 2004. — [4]

[113] A. Schmidt, K. G. Siebert, D. Köster, and C.-J. Heine. *ALBERTA 3.0: Technical Manual*. Mar. 2012. URL: http://www.mathematik.uni-stuttgart.de/fak8/ians/lehrstuhl/nmh/downloads/alberta/alberta-man.pdf (visited on 06/28/2013). — [4]

[114] *The Boost C++ libraries*. URL: http://www.boost.org. — [6, 36, 71, 115]

[115] *Trilinos Website*. URL: http://trilinos.sandia.gov/index.html. — [5, 18]

[116] B. Uekermann, H.-J. Bungartz, B. Gatzhammer, and M. Mehl. "A Parallel, Black-Box Coupling for Fluid-Structure Interaction". In: *Computational Methods for Coupled Problems in Science and Engineering, COUPLED PROBLEMS 2013*. Ed. by S. Idelsohn, M. Papadrakakis, and B. Schrefler. 2013. — [4]

[117] *UG web site*. URL: http://atlas.gcsc.uni-frankfurt.de/~ug/ (visited on 06/29/2013). — [4]

[118] E. Unruh. "Prime number computation". ANSI X3J16-94-0075/ISO WG21-462. 1994. — [30, 31]

[119] D. Vandevoorde and N. M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2002. — [31]

[120] T. L. Veldhuizen. "Using C++ Template Metaprograms". In: *C++ Report* 7.4 (1995), pp. 36–43. — [30, 31]

[121] J. Walter, M. Koch, G. Winkler, and D. Bellot. *The uBLAS Library*. 2000–2010. URL: http://www.boost.org/libs/numeric/ublas/. — [6]

[122] T. Wick. "Coupling of fully Eulerian and arbitrary Lagrangian–Eulerian methods for fluid-structure interaction computations". In: *Computational Mechanics* 52.5 (2013), pp. 1113–1124. DOI: 10.1007/s00466-013-0866-3. — [2]

[123] B. Wohlmuth. *Discretization Techniques and Iterative Solvers Based on Domain Decomposition*. Lectures Notes in Computational Science and Engineering 17. Springer, Heidelberg, 2001. — [2, 132]

[124] M. Wohlmuth. "Modeling and Simulation of Solid-State Laser Resonators Using a Dynamic Multimode Analysis (DMA)". PhD thesis. Universität Erlangen-Nürnberg, 2012. — [129]

[125] H. Yang. "Partitioned solvers for the fluid-structure interaction problems with a nearly incompressible elasticity model". In: *Computing and Visualization in Science* 14.5 (2011), pp. 227–247. DOI: 10.1007/s00791-012-0177-9. — [2]

Acknowledgments

First and foremost, I was very fortunate to have Prof. Dr. Peter Bastian as my advisor. He left me the freedom to pursue my work in the way I preferred to, but was always available when I got stuck; his experience and knowledge were invaluable to achieve the right balance for many of the fundamental design decisions in my research project. I would also like to express my gratitude for the encouragement and the trust he placed in me throughout these years. I am also indebted to my co-examiner Prof. Dr. Thomas Ertl, who not only examined this thesis, but also provided me with an academic "home" in his computer visualization group during my time in Stuttgart. Without this hospitality, that time would have been rather lonely. Finally, I would also like to thank Prof. Dr. Christian Engwer, with whom I had long and very helpful discussions about the intricacies of DOF reordering.

I am also very grateful to Prof. Hans-Petter Langtangen, who gave me the opportunity to stay at the SIMULA research laboratory in Oslo for one summer and to extend my horizon by looking at how finite elements are approached by other software projects. At this point, I would also like to thank the SIMTECH cluster of excellence, who not only paid my wages during my time in Stuttgart, but also financed the stay in Norway.

Last but not least, a big thank you to all the users of my software, who have not only provided me with valuable input, but seeing how they were able to do fascinating science with its help has also given me the feeling that there is a real purpose to the work I have been doing over the last years.

During my work on this thesis, I was part of not one, but two working groups full of wonderful colleagues. Thanks to everybody in the VIS group in Stuttgart and in particular to Markus Üffinger, the Höferlin brothers, Harald Sanftmann, Martin Falk, Michael Wörner and whoever else was around for the many discussions about life, the universe and everything else that had not disappeared on the count of three – I miss the great coffee rounds after lunch! A special thanks also to Martin for his formidable LaTeX template. A year ago, I moved to Heidelberg to become a "proper" member of the group of Prof. Bastian. For all the nice discussions, time together, shared coffee, trips to the shop and all your help I am grateful to Pavel Hron (and his ever welcoming flatmates), Jurgis Pods, Rebecca Neumann, Dominic Kempf and all the other members of the AG Wissenschaftliches Rechnen.

I would like to thank my parents Hannelore and Michael for their continued and never waivering support throughout my studies and this thesis and all my flatmates Carmen, Tanja, Leni, Daniela and Astrid for their patience during the last years, when my mood was often darkened by the work on this thesis. Finally, I would like to thank Kristina for her warmth and moral support and for her proofreading, and Anja, for showing me again that life is more than just obligations and giving me the faith in myself that I needed to make it through the last few stressful months.

Erklärung

Hiermit versichere ich, Steffen Müthing, dass ich die vorliegende Arbeit selbständig angefertigt habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe. Ich erkläre außerdem, dass die von mir vorgelegte Dissertation bisher nicht im In- oder Ausland in dieser oder ähnlicher Form in einem anderen Promotionsverfahren vorgelegt wurde.

Stuttgart, 15. Dezember 2014

(Steffen Müthing)