Technische Universität München

Lehrstuhl für Informatik mit Schwerpunkt
Wissenschaftliches Rechnen

Immersed Boundary Methods within a PDE Toolbox on Distributed Memory Systems

Janos Benk

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Georg Carle

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Hans-Joachim Bungartz
2. Prof. George Biros, Ph.D.
   University of Texas Austin/USA
   (nur schriftliche Beurteilung)
3. Univ.-Prof. Dr. Christoph Zenger, i.R.

Die Dissertation wurde am 12.04.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03.08.2012 angenommen.

Abstract

Advanced simulation of complex physical systems governed by partial differential equations (PDE) poses significant computational challenges that require a collection of sophisticated numerical methods. One of the main challenges is the representation of complex boundaries and domains together with the respective boundary conditions (BC). In the classical way, this challenge is tackled by a costly mesh generation process, that becomes a significant computational bottleneck especially on distributed memory systems. The time to solution is a crucial factor for modern PDE software development. Hence, combining new numerical methods into a user-friendly PDE toolbox that also facilitates parallel simulations is a significant algorithmic and software design challenge.

This thesis describes contributions to the development of various complex boundary representations in the form the Immersed Boundary (IB) methods within the frame of the PDE toolbox Sundance, a package within the Trilinos project. The IB methods use a memory- and cache-efficient structured mesh in combination with special methods to impose the BCs on complex boundaries. We extended Sundance with parallel structured mesh implementation, while general cut-cell and boundary integral methods were developed in the frame of Sundance, allowing the implementation and parallel computation of various IB methods in this toolbox environment. The one particular IB method in our focus is Nitsche's method for flow simulation that facilitates moving boundaries even for a fixed mesh approach, and significantly simplifies the obstacle representation in the flow field.

To demonstrate the capabilities of our IB approach and Sundance implementation we computed various benchmark scenarios in 2D and 3D settings. The presented results of the strong scaling study show the scalability on distributed memory systems. This work, thus, is an important step towards IB methods in a PDE toolbox context, capable of distributed memory simulation.

Zusammenfassung

Simulationen von komplexen physikalischen Systemen, die mit partiellen Differentialgleichungen (PDE) beschrieben werden, stellen große Herausforderungen an heutige Rechner und erfordern daher effiziente numerische Methoden. Eine der wichtigsten Herausforderungen ist die Darstellung komplexer Ränder und Gebiete zusammen mit den jeweiligen Randbedingungen. Im klassischen Ansatz wird diese Aufgabe durch einen rechenintensiven Gittergenerierungsprozess bewältigt, welcher vor allem bei parallelen Simulation und insbesondere auf Distributed-Memory-Systemen einen erheblichen Engpass darstellt. Die Time-to-Solution ist ein entscheidender Faktor für die moderne PDE-Software-Entwicklung. Die dafür notwendige Kombination neuer und effizienter numerischer Methoden mit einer benutzerfreundlichen PDE-Toolbox, die auch parallele Simulationen ermöglicht, stellt bedeutende Anforderungen an die zugrundeliegende Algorithmik und das Software-Design.

Diese Arbeit beschreibt wichtige Beiträge zur Entwicklung verschiedener komplexer Randdarstellungen mit Hilfe der Immersed-Boundary (IB) Methoden, die in der PDE-Toolbox *Sundance*, einem Paket im *Trilinos* Projekt, implementiert wurden. IB-Methoden verwenden ein speicher- und cacheeffizientes strukturiertes Gitter in Kombination mit speziellen Methoden, die die Randbedingungen auf komplexen Geometrien ermöglichen. Wir haben die PDE-Toolbox um parallele strukturierte Gitter erweitert und gleichzeitig allgemeine Cut-Cell- und Randintegral-Methoden entwickelt. Diese allgemeinen Methoden ermöglichen die Implementierung und anschließend die parallele Simulation von schwach formulierten IB-Methoden in der *Sundance*-Toolbox-Umgebung. In dieser Arbeit wird eine bestimmte IB-Methode, die Nitsche-Methode, für Strömungssimulationen implementiert, die bewegte Ränder für ein fixiertes Gitter ermöglicht. Damit vereinfacht sich die genaue Hindernis-Darstellung in einem Strömungsfeld deutlich.

Um das Potenzial unseres IB-Ansatzes in der Sundance-Implementierung zu demonstrieren, berechnen wir verschiedene Benchmark-Szenarien in 2D und 3D, die unseren Ansatz verifizieren. Die erzielte starke Skalierbarkeit zeigt die parallele Effizienz des verwendeten Ansatzes auf Distributed-Memory-Systemen. Diese Arbeit stellt einen wichtigen Schritt in Richtung allgemein anwendbarer IB-Methoden in einer PDE-Toolbox für Distributed-Memory-Systeme dar.

Contents

1. Introduction **7**

2. Finite Element Basics **11**
2.1. Fundamentals from Functional Analysis 11
2.2. Finite Element Discretization 17

3. Governing Equations in the Applications **31**
3.1. Fluid Model 31
3.2. Structure Model 35
3.3. Fluid-Structure Interaction 38

4. Cartesian Meshes and Immersed Boundary Methods **41**
4.1. Cartesian Meshes 41
4.1.1. Tree-Structured Cartesian Meshes 42
4.1.2. Cartesian Mesh Traversal and Domain Decomposition 45
4.1.3. Geometry and Boundary Representation 46
4.2. Immersed Boundary Methods 48
4.2.1. Overview 48
4.2.2. Penalty Method 51
4.2.3. Finite Cell Method 54
4.2.4. Lagrange Multiplier Method 55
4.2.5. Nitsche's Method 57

5. Sundance PDE Toolbox Introduction **65**
5.1. Structure of the Sundance PDE Toolbox 65
5.1.1. Problem Formulation 66
5.1.2. Matrix Assembly 75
5.1.3. Solvers 82
5.1.4. Visualization 83
5.2. Overview of Open-source FEM-based PDE Toolboxes 84

6. Parallel Adaptive Cartesian Meshes in Sundance **87**
6.1. Quad and Brick Elements in Sundance 87
6.2. The Pre-fill Element Transformation for Hanging Degrees of Freedom . . 88
6.3. Sundance Mesh Interface Extensions 95
6.4. Degree of Freedom Map Extensions for Hanging DoFs 96

6.5. Parallel Adaptive Cartesian Mesh Implementations in Sundance 98
6.5.1. Mesh Storage and Runtime Comparison 101

7. Fluid Flow with Nitsche's Method 105
7.1. Boundary Geometry Representation 105
7.1.1. Geometry Interface and Analytical Geometry Representations . . 106
7.1.2. Polygons as Two-dimensional Geometry 110
7.1.3. Triangle Surfaces as Three-dimensional Geometry 113
7.2. Cut-Cell Quadrature 118
7.2.1. 2D Cut-Cell Integration Method 120
7.2.2. 3D Cut-Cell Integration Method 123
7.2.3. Cut-Cell Integration Methods in Sundance 126
7.3. Curve and Surface Integrals 127
7.3.1. 2D Curve Integration 128
7.3.2. 3D Surface Integration 129
7.3.3. Curve and Surface Integral Implementations in Sundance 130
7.4. Fluid Flow Benchmark Results 132
7.4.1. 2D Benchmark Results 133
7.4.2. 3D Benchmark Results 137

8. Fluid-Structure Interaction with Nitsche's Method 139
8.1. Moving Geometries with Nitsche's Method in 2D and 3D 139
8.2. Partitioned Fluid-Structure Interaction 142
8.2.1. Stationary FSI 142
8.2.2. Partitioned and Transient FSI with Explicit and Implicit Time Coupling 143
8.2.3. Implementational Requirements in Sundance 145
8.3. 2D Results 147
8.3.1. 2D Stationary Results 148
8.3.2. 2D Transient Results 152
8.4. 3D Results 155
8.4.1. 3D Stationary Coupling Results 156
8.4.2. 3D Explicit Coupling Results 158

9. Porous Media Simulation with the Stokes-Brinkman Model 161
9.1. The Governing Equation and the Geometry Model 161
9.2. Computational Results in 2D and 3D 163
9.2.1. 2D Results 164
9.2.2. 3D Results 165
9.3. Strong Scaling Results of the 3D Parallel Computations 166
9.3.1. Results with Q_1Q_1 Elements 167
9.3.2. Results with Q_2Q_1 Elements 169

10. Summary and Outlook **171**
10.1. Summary 171
10.2. Outlook 172

A. Appendix **173**
A.1. Notations for Structural Mechanics 173
A.2. Nitsche's Method Derivation for the Poisson Equation 175
A.3. Nitsche's Method Derivation for the Navier-Stokes Equations 177
A.4. Sundance Code for the Navier-Stokes Equations with Nitsche's Method . 180
A.5. Sundance Code for Static Partitioned FSI Computations 183

1. Introduction

Classical scientific research consisted of only two pillars. The first one is the theoretical model that the scientist assumes for a given physical phenomenon, and the second pillar is the experiment or observation, where the validity of the proposed theoretical model could be proved or disproved. The theoretical mathematical models are often complex and analytical solutions are rarely available in particular for realistic scenarios. This limits the capability to model and to verify a given model for a complex physical phenomenon. Furthermore, experiments and observations might be impossible, too expensive, or too dangerous to make in a real-world setting.

For these reasons, numerical methods are employed in a combination with hardware-efficient algorithms, in order to efficiently compute the solutions of complex models on modern computing architectures. The resulting solutions are then further analyzed or compared to measured data in order to gain knowledge from the process. The discipline that incorporates these types of approaches is called *scientific computing*, representing the third pillar of scientific research. Scientific computing already helped scientists to make advances in many different areas (nuclear fusion, astronomy, quantum chemistry, e.g.). Furthermore, computational science is gaining more and more importance not only in science but also in engineering, where the goal is not just insight, but a further improvement of a product or a speed up of development phases.

A significant portion of complex physical systems are governed by partial differential equations (PDE), for which analytical solutions are only known in very simple cases. The first step towards a numerical solution is to use a spatial discretization. One of the most common and general ways to discretize a PDE is the finite element method (FEM). This discretization method fits well to a general software structure that enables the modular construction of a PDE toolbox, where each component is replaceable. Such a modular structure of a PDE toolbox allows for the reusability of developed code, such that a given PDE problem can be computed with a chosen discretization, quadrature method, and solver. Our goal in this thesis is to develop a combination of a memory-saving structured adaptive mesh implementation with a sophisticated accurate treatment of boundary condition. This approach is then integrated into the frame of a PDE toolbox, which improves the usability and the user friendliness of our implementation. We choose the FEM-based PDE toolbox Sundance [60, 61] that has a high-end problem description language and also allows for the efficient implementation of various PDE models. Sundance also has the built-in capability to run in parallel on distributed memory systems that is nowadays a 'must-have' requirement for simulation softwares.

One of the main challenges in solving PDEs numerically is to represent the boundary of the computational domain accurately and to impose a given boundary condition (BC) on it. The classical way is to create a mesh that approximates well the boundary with its facets. This typically leads to unstructured meshes, for which both the generation overhead and the memory requirements are high. This hold in particular for complex and moving geometries. An alternative approach that we follow in this thesis, is to use immersed boundary (IB) methods, where the boundary is represented by a different entity. Therefore, IB methods allow for the usage of a structured and computationally cheaper mesh. The task here is to impose a given BC on the immersed boundaries, which do not coincide with the mesh's facets. For this purpose, we investigate several IB approaches stated in a weak form and apply them to various applications. One of these methods is Nitsche's method [69], which we employ for the first time for Navier-Stokes equations in an IB setting. This method proves to be not only consistent on the boundary but also efficient for complex domains.

Concluding and completing the above mentioned challenges, we enlist the following aspects of the FEM-based PDE simulation that we address in this thesis and represent the major contributions of this work:

- **PDE Toolbox with Immersed Boundary Capabilities**: We only consider IB methods that can be formulated in a weak form, allowing for the implementation in a FEM-based PDE toolbox. For this purpose, we develop IB capabilities within the PDE toolbox that consist of various efficient cut-cell and boundary integration methods. IB methods further require an explicit geometry representation, since the boundary is not represented by the mesh's facets. Such a boundary geometry representation is also a challenge that needs to be tackled in this context.

- **Immersed Boundary Methods**: With the developed IB capabilities, we propose to investigate and develop various IB methods. We are mainly interested in approaches that are capable to weakly impose Dirichlet BCs for incompressible fluid simulation, not only on fixed boundaries but also on moving geometries, while the underlying mesh remains fixed.

- **Adaptive Cartesian Mesh**: In the IB context, it is reasonable to use a structured mesh that can be created and refined in a simple way and further require considerably less storage than a comparable unstructured mesh. Such a structured mesh is the adaptive Cartesian mesh that allows for space-filling curve based domain decomposition for parallel simulation on distributed memory systems. The implementation of such a parallel mesh within the frame of a PDE toolbox is one of the challenges that we tackle in this thesis.

- **PDE Toolbox Integration**: The reusability of modern computational software, typically containing various numerical methods, usually represents a significant problem. In some cases, classical SE design approaches are avoided in order to maximize the performance of the computations. Such approaches are justified

only for a few performance-critical sections of a simulation code. A similar idea is also suggested by Knuth: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil".[1] Therefore, our goal is to implement all the proposed methods in Sundance, such that any user can freely access them, while the overall performance of the implemented methods is not compromised.

- **Parallel Implementation**: Sundance allows by design a distributed memory execution. While we extend this toolbox with IB capabilities, we also have to make sure that all algorithms and data structures scale well on parallel systems. Our goal is to develop IB capabilities that are not only usable for the toolbox users, but also deployable in parallel simulations with good scaling properties.

- **Multi-physics**: The simulation of multi-physics problems often poses implementational and computational challenges for a given scientific software. Sundance allows for the straightforward declaration of various PDEs, which model different physical phenomena. The volume coupling of two models can be realized in the weak form of the coupled problem, whereas an interface coupling implies more implementational and numerical issues. As a multi-physics problem, we consider several FSI scenarios. Our goal is here to develop general interface coupling capabilities that allow for the simulation of interface coupled multi-physics problems within Sundance.

This thesis is structured in 10 chapters and one appendix. The main research contributions from the Phd project documented in this thesis are presented in Chapter 6, Chapter 7, Chapter 8, and Chapter 9. We start with the general introduction of FEM by briefly presenting the mathematical background of the method in Chapter 2. In Chapter 3, we present the governing equations in the applications that we compute in this thesis. We show the fluid's model, the structure's PDE, and the mathematical formulation of an FSI problem. Chapter 4 starts with an overview of the adaptive Cartesian mesh that can be used efficiently in combination with IB methods for parallel simulations. Furthermore, we give an overview of IB methods where the focus is on the methods that impose the BCs weakly. We chose Sundance as a baseline for our implementation. The main architecture and features of this PDE toolbox are presented in Chapter 5. This chapter closes with an overview of existing open-source, FEM-based PDE toolboxes that are currently available. In the following chapters of the thesis, we introduce the developed features and methods for IB computations. Chapter 6 presents the extensions of Sundance with rectangular elements, that in case of adaptive Cartesian meshes require the handling of irregularities caused by so-called hanging nodes. We show, how we extended Sundance with such meshes and how the irregularities are resolved in a general and user-transparent approach. In Chapter 7, we introduce the developed general IB capabilities within Sundance, which consist of cut-cell and boundary

[1] Knuth, Donald. Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.

integral methods and require an explicit geometry representation in 2D and 3D. The boundary geometry representation with the developed IB features is deployed here for Nitsche's method to compute 2D and 3D flow benchmarks. Once Nitsche's method is verified for stationary obstacles, we extend this method for moving boundaries within fluids in Chapter 8. With this extension, we compute 2D and 3D FSI problems, where the 2D benchmark values verify the correctness of our approach. Last, in Chapter 9, we employ a different type of IB method to compute the permeability of a porous medium, namely the volume penalty method. We close this thesis with a summary of the obtained results and with an outlook on future research directions.

Acknowledgments

This research work has been accomplished with the strong support of the Munich Centre of Advanced Computing (MAC[2]) and of the Chair of Scientific Computing in Computer Science Faculty[3] at the Technische Universität München. The author also acknowledges the support of IGSSE[4] for his two month stay at the Texas Tech University (TTU).

I want to thank my supervisor Prof. Dr. Hans-Joachim Bungartz for offering me the opportunity to do my doctoral thesis on such an fascinating research topic. He always offered unconditional support for my research work and gave me new impulses and ideas when I got stuck. I also want to thank for the opportunity to participate in other research projects that helped me to broaden my knowledge. Further, I want to thank Prof. Dr. Michael Ulbrich for his support within the frame of the MAC-B7 project, especially for the few but very crucial Sundance code debugging and mathematical advices. Further, I want to say many many thanks to Prof. Robert Kirby, Phd and to Prof. Kevin Long, Phd. They supported me to become a Sundance developer. During my stay at TTU, they helped me with many ideas and advices to integrate new features into Sundance. Special thanks go also to Dr. Miriam Mehl for her support and help during my research work and for her feedback on this manuscript. Further, I also want to say many thanks to the colleagues who helped and supported me in my research and in the writing of this thesis.

Last but not least, I want to thank my family for their unconditional support that made me able to do my Phd studies.

[2]www.mac.tum.de
[3]http://www5.in.tum.de
[4]http://www.igsse.tum.de/

2. Finite Element Basics

The Finite Element Method is one of the most widespread and general method to discretize a PDE. The generality of the FEM allows for the implementation of PDE toolboxes such as Sundance, which is in the focus of this thesis. At the same time, FEM is the mathematically most founded discretization method. In this chapter, we enlist the necessary functional analysis fundamentals for the finite element method, while the main focus here is to introduce the FEM. Besides the theoretical information, the introduced terms and theory play an important role in Chapter 5 and Chapter 6, where the different components of the Sundance toolbox and its weak form based syntax are described. The FEM represents the first step in a toolbox approach to the discretization of the PDE and restricting the solution function to a discrete finite dimensional space. Before we make the step to a finite space, we have to consider several aspects of the functional spaces in their initial infinite dimensions.

In the first part, we enlist the necessary functional analysis basics of the finite element method. In the second part of the chapter, we introduce the FEM with the Ritz-Galerkin approach, which is the most common form of FEM discretization, especially in a toolbox context.

2.1. Fundamentals from Functional Analysis

In general, a PDE problem can be seen as the strong form of $Lu = f$, where u is the solution function, L a differential operator, and f a given function. The solution function is contained in a function space with different characteristics. In the following, we enlist the theory related to functional spaces. The notations for functional spaces defined here are used in the following chapters, when we define a concrete PDE problem. We start with the different types of functional spaces and their properties, which are crucial for the finite element theory. For more detailed insights and for the proofs of the theorems, we refer to [23, 21, 50].

In the following, we denote function spaces by a capital letter V and assign to such a space a norm. The norm $\|\cdot\|$ is a mapping (function) $\|\cdot\| : V \longrightarrow \mathbb{R}^+$ from the elements of the space V to positive real numbers $[0, \infty[$ with the following four properties: (1) $\|v\| \geq 0,\ \forall v \in V$, (2) $\|v\| = 0 \Leftrightarrow v = 0$, (3) $\|c \cdot v\| = |c| \, \|v\|,\ \forall c \in \mathbb{R},\ v \in V$, (4) $\|w + v\| \leq \|w\| + \|v\|$, for $v, w \in V$. The norm is important in order to have a

given distance metric between the elements of the function space. Such a metric is called *complete* if every Cauchy sequence $\{v_i\}$ in V has a limit $v \in V$. This means, $\|v - v_j\| \longrightarrow 0$ while $j \longrightarrow \infty$. Using this complete metric in a linear space V (where additivity and multiplicativity holds) defines the first type of space.

Definition 2.1.1 *A normed linear space, denoted by $(V, \|\cdot\|)$, is called a **Banach space**, if the metric defined by the norm $\|\cdot\|$ is **complete**.*

We further introduce the **dual space** B' to a given Banach space B. The dual space B' includes all the linear functionals $F : B \longrightarrow \mathbb{R},\ F(v + aw) = F(v) + aF(w),\ \forall v, w \in B,\ a \in \mathbb{R}$, with the associated norm

$$\|F\|_{B'} := \sup_{v \in B, v \neq 0} \frac{F(v)}{\|v\|_B}.$$

The functionals in the dual space play an important role in the right-hand side of the PDEs in the weak form, where the test space is the input for a given right-hand side functional F, which is element of the dual space.

In the following, we define a common metric that results in *Lebesgue spaces*. We consider a real valued function f on a subset Ω of $\mathbb{R}^n$. Then, the Lebesgue integral of f is defined as $\int_\Omega f(x)dx$, where dx denotes the Lebesgue measure. With this integral, the following metric can be defined

$$\|f\|_{L^p(\Omega)} := \left(\int_\Omega |f(x)|^p \, dx \right)^{\frac{1}{p}},$$

where $1 \leq p < \infty$, and for the case $p = \infty$

$$\|f\|_{L^\infty(\Omega)} := \text{ess sup}\{|f(x)| \ : \ x \in \Omega\}.$$

In short notation, this norm is denoted by $\|\cdot\|_p := \|\cdot\|_{L^p(\Omega)}$, $1 \leq p \leq \infty$. Using this metric, the *Lebesgue space* $L^p(\Omega)$ is defined as

$$L^p(\Omega) := \{f \ : \ \|f\|_{L^p(\Omega)} < \infty\}. \tag{2.1}$$

It is straight forward to derive that the Lebesgue space $L^p(\Omega)$ with $1 \leq p \leq \infty$ is also a Banach space. Using the Lebesgue measure and integral in the definitions above has important aspects. Two functions f and g are equal in the Lebesgue norm, if they have the same values almost everywhere. If they differ only in subsets (e.g., pointwise in 1D) with Lebesgue measure zero, they are still equal in the sense that $\|f - g\|_{L^p_{(\Omega)}} = 0$. This property allows this integral to be defined also for improper integrals, crucial for the completeness of the induced norm.

Next, we turn our attention to the definition of the *weak derivative*, which is defined through partial integration. In contrast to the classical calculus's pointwise view of the

derivative, the weak derivative is determined by the global behavior of the function. In the first step, we define the multi-index vector α, with n non-negative integers α_i. The length (Manhattan norm) of this vector is given as

$$|\alpha| := \sum_{i=0}^{n} \alpha_i.$$

Using this multi-index vector, we denote the partial derivative of a given function f as $D^\alpha f$, $\left(\frac{\partial}{\partial x}\right)^\alpha$ or simply f^α. In a detailed notation, this partial derivative has the form

$$\left(\frac{\partial}{\partial x_1}\right)^{\alpha_1} \dots \left(\frac{\partial}{\partial x_n}\right)^{\alpha_n} f.$$

With the presented notation, we arrive to the definition of the weak derivatives, where we use the notion of locally integrable function, which we denote with $L^2(\Omega)$.

Definition 2.1.2 *We define $g \in L^2(\Omega)$ as the **weak derivative** D_w^α of $f \in L^2(\Omega)$, if the following condition holds:*

$$\int_\Omega g(x)\varphi(x)dx = (-1)^{|\alpha|} \int_\Omega f(x) D^\alpha \varphi(x)dx \;\; \forall \varphi \in Z_0(\Omega).$$

$Z(\Omega)_0$ defines all the functions in Ω, which have a compact support.[1]

All $\varphi \in Z_0(\Omega)$ vanish at the boundary $\partial\Omega$. Thus, the integration by parts in the weak derivative definition Def. 2.1.2 results in vanishing boundary integrals. If the function f is $C^{|\alpha|}$-continuous, its weak derivative also exists and further we can write that $D_w^\alpha f = D^\alpha f = g$. The existence of the weak derivatives of a function f is the prerequisite for the definition of the Sobolev spaces:

Definition 2.1.3 *Let k be a positive nonzero integer and $f \in L^2(\Omega)$. Assuming that the weak derivatives exist in the sense of Def. 2.1.2, the **Sobolev norm** is defined as*

$$\|f\|_{W_p^k(\Omega)} := \left(\sum_{|\alpha| \le k} \|D_w^\alpha f\|_{L^p(\Omega)}^p \right)^{\frac{1}{p}},$$

if $1 \le p < \infty$, and in case $p = \infty$

$$\|f\|_{W_\infty^k(\Omega)} := \max_{|\alpha| \le k} \|D_w^\alpha f\|_{L^\infty(\Omega)}$$

*In both cases, the **Sobolev spaces** are defined as*

$$W_p^k(\Omega) := \{ f \in L^2(\Omega) \; : \; \|f\|_{W_p^k(\Omega)} < \infty \}$$

[1]The support is the domain $X \subset \Omega$ where the function is nonzero.

From Def. 2.1.3 follows that all elements of the Sobolev space must have a bounded weak derivative up to an order k. This requirement is important later for the finite element discretization. In the following, we also denote $W_p^2(\Omega)$ with $H^p(\Omega)$.[2]

Functional spaces with associated bilinear forms are the next step towards the weak formulation of the finite element method. A bilinear form takes two elements as input from two functional spaces and maps them to real values $b(\cdot,\cdot) : V \times V \longrightarrow \mathbb{R}$, such that it is a linear map.[3] If this bilinear form is symmetric, then $b(v,w) = b(w,v)$ for $\forall v, w \in V$. One example of such a bilinear form is $b(w,v) = \int_\Omega v(x)\, w(x)\, dx$, which is also symmetric. Further, we define an additional category of such operators. A symmetric bilinear operator is an **inner product** on space V, if the following conditions are satisfied: $b(v,v) \geq 0\ \forall v \in V$ and $b(v,v) = 0 \Leftrightarrow v = 0$. An **inner product** $b(\cdot,\cdot)$ together with a space V is defined as **inner-product space**, and is denoted as $(V, b(\cdot,\cdot))$. Such an inner-product space is $\left(L^2(\Omega), b(v,w) = \int_\Omega w(x)\, v(x)\, dx\right)$, for which the above defined conditions hold. It is important to note that this inner product also induces a norm on V, where we can write $\|v\| = \sqrt{b(v,v)}$. If this metric is complete, then this triple defines the next space.

Definition 2.1.4 *Given the inner product space $(V, b(\cdot,\cdot))$, and the associated normed space $(V, \|\cdot\|)$. If this normed space is complete then $(V, b(\cdot,\cdot))$ is called a **Hilbert space***

Further, we define a closed linear subspace S in V, which means that $\forall v, w \in S,\ \alpha \in \mathbb{R} \Longrightarrow v + \alpha w \in S$. If such a subspace S exist then $(S, b(\cdot,\cdot))$ is also a Hilbert space. Following the logic in [23], we further define two characteristics of a bilinear form.

Definition 2.1.5 *A bilinear form $b(\cdot,\cdot)$ on a normed linear space H is **bounded** (or **continuous**) if there is a positive constant $C < \infty$ such that*

$$|b(v,w)| \leq C \|v\|_H \|w\|_H, \quad \forall v, w \in H$$

*and is **coercive** on $S \subset H$, if there exist an $\alpha > 0$ such that*

$$b(v,v) \geq \alpha \|v\|_H^2, \quad v \in V.$$

In the following, we denote the inner-product by simply $(\cdot,\cdot)$, and the inner product space with the linear space V as $(V, (\cdot,\cdot))$. At this stage, we have the necessary background to state the PDE problem given a bilinear form $a(u,v),\ u, v \in V$. Given a linear functional $F \in V'$ find $u \in V$ such that

$$a(u,v) = F(v), \quad \forall v \in V. \tag{2.2}$$

[2] In other publications $H^p(\Omega)$ denotes the Hilbert space (introduced in the next section) with the inner-product $(u,v) := \int_\Omega uv\, dx$

[3] One has to mention here, that in general cases the bilinear form could use different functional spaces, e.g., $b(\cdot,\cdot) : V \times W \longrightarrow \mathbb{R}$

Here, we can make the connection to the differential operator L, which represents the PDE in the strong form.[4] The weak form can have the following initial form

$$\int_\Omega Lu\, vdx = F(v), \quad \forall v \in V,$$

with $a(u,v) = \int_\Omega Lu\, v\, dx$, where the second order derivatives are later transformed by partial integrations. We consider the concrete example with $Lu = -\Delta u$ and $F(v) = (f,v)$ as a simple linear functional

$$\int_\Omega -\Delta uvdx = \int_\Omega \nabla u \nabla vdx - \oint_{\partial\Omega} (\nabla u\, \mathbf{n})\ vdc = \int_\Omega fvdx, \quad \forall v \in V,$$

where $\mathbf{n}$ is the normal vector pointing outwards of the domain Ω. At this point, we introduce a notation that is commonly used in literature: We rewrite the equation above as

$$(\nabla u, \nabla v)_\Omega - ((\nabla u\, \mathbf{n}), v)_{\partial\Omega} = \langle f, v\rangle_\Omega,$$

where $(\cdot,\cdot)$ and $\langle f,\cdot\rangle$ represent the corresponding integrals. Since v is zero at the boundary $\partial\Omega$, the variational form of our example simplifies to

$$\int_\Omega \nabla u \nabla vdx = \int_\Omega fvdx, \quad \forall v \in V. \tag{2.3}$$

Several questions arise at this point. Does this problem have any solutions? If yes, then is this solution unique? We answer these question in a more general case, where the bilinear operator $a(u,v)$ is not necessarily symmetric.

Theorem 2.1.1 ***(Lax-Milgram)*** *Given a Hilbert space $(V,(\cdot,\cdot))$, a continuous, coercive bilinear form $a(\cdot,\cdot)$, and a linear continuous functional $F \in V'$. There is a unique $u \in V$ such that*

$$a(u,v) = F(v), \quad \forall v \in V.$$

This function u is also the unique solution of the minimization problem

$$J(u) := \frac{1}{2}a(u,u) - F(u) \longrightarrow \min!$$

Theorem. 2.1.1 assures the existence and uniqueness of the solution of the weak form such as (2.2). This weak form is usually transformed by one ore more partial integrations, as we showed for the Poisson equation in (2.3), and this form is also called the **weak formulation** of the PDE problem. The adjective *weak* suggest that the requirements for the solution functions has been *weakened.* In the case of the Poisson equation $-\Delta u = f$,

[4] This connection is also shown through the Riesz Representation Theorem [23], where $L_u(v)$ is the inner-product on the Hilbert space and is an element of the dual space.

the requirement is that $u \in C^2(\Omega)$, but in the weak form (2.3), we can state that $u \in H^1(\Omega)$. This means that u in the weak form is required to have only up to the first order bounded weak derivatives, compared to the twice differentiable assumption in the strong form. This *weaker* restriction also means that in lower dimensional elements (e.g., pointwise in 1D), which is not measurable in the Lebesgue measure, u might have even undefined first derivative. These properties allow the finite element method to compute the Poisson problem in the one-dimensional case with only piecewise linear basis functions, since they are in the Sobolev space $H^1(\Omega)$.

At this stage, we also introduce the notations related the weak form. The function $v \in V$ in Theorem. 2.1.1 is called **test function**, whereas the function $u \in V$ represents the **unknown** or **ansatz function**. $\Omega \subset \mathbb{R}^d$ represents the computational domain of the problem. Such a weak form is the main requisite for the finite element discretization, and it can be used straight forward to create a linear system of equations by discrete ansatz and test spaces (see Section 2.2).

In order to have a well-posed problem, in the PDE context, often boundary conditions are required. At this point, the question arises how the boundary conditions play a role in the weak formulation of the problem and how are they imposed at the boundary. The answer to this question is the topic of the remaining part of this section.

To impose boundary conditions the boundary $\partial\Omega$ of Ω has to be a Lipschitz boundary ($\longrightarrow \Omega$ is a Lipschitz domain). This practically means that the boundary is sufficiently regular and continuous. If this condition is met, then the **Trace theorem**[5] gives us an upper limit of the function norm measured in $\|\cdot\|_{L_p(\partial\Omega)}$ at the boundary. Once the function value is bounded on the boundary $\partial\Omega$, the error between the imposed and actual values can be measured. This is assured, if the boundary is regular and continuous, which is the case in most practical applications.

In the next step, we investigate the case of Dirichlet boundary conditions, $u|_{\partial\Omega} = g_D$. This condition binds the value of the unknown function u to a given function value g. This way, the unknown function values are known on $\partial\Omega$ and is not an unknown value at $\partial\Omega$ in the weak formulation. For this simple reason, test functions $v \in H_0^1(\Omega)$ with compact support[6] are employed. With such v, the integration by parts of the Poisson equation $\int_\Omega -\Delta uv dx$ results only in a volume integral term $\int_\Omega \nabla u \nabla v dx$ and the boundary integral $\oint_{\partial\Omega} -(\nabla u \mathbf{n}) v dc$ vanishes. This rule is valid also for the a general PDE's weak form derivation. The main question still remains how to enforce $u|_{\partial\Omega} = g_D$. This enforcement is usually done at the discrete level as will be described in the next section. However, on the continuous level one can also state the boundary condition in the weak form is

$$\int_{\partial\Omega} (u - g_D) w dx = 0, \quad \forall w \in L^2(\partial\Omega), \tag{2.4}$$

where w is a test function existing only on the boundary.

[5] For more details we refer to [21].
[6] v has zero values on the boundary

The next type of boundary condition is the Neumann condition, which states $\frac{\partial u}{\partial \mathbf{n}}|_{\partial\Omega} = g_N$. In this case, the values of the unknown (or ansatz) function are not fixed on the boundary, but only their gradient in the normal direction. Thus, u remains unknown on $\partial\Omega$. Hence, the test function v should not vanish on $\partial\Omega$:

$$\int_\Omega -\Delta u v dx = \int_\Omega \nabla u \nabla v dx - \oint_{\partial\Omega} (\nabla u \mathbf{n})\, v dc = \int_\Omega f v dx.$$

The boundary integral term contains the normal derivative of u, which we can replace by g_N. By this step, we already arrive at the final weak form of the Poisson problem

$$\int_\Omega \nabla u \nabla v dx - \oint_{\partial\Omega} g_N v dc = \int_\Omega f v dx, \tag{2.5}$$

which includes the Neumann boundary condition. This method of imposing Neumann boundary condition is also valid for a general PDE problem (that contains diffusion). Neglecting the boundary integral implicitly imposes zero Neumann boundary condition in the weak form.

In case of mixed boundary conditions on $\Gamma = \partial\Omega$, where $\Gamma = \Gamma_N \cup \Gamma_D$ and $\Gamma_N \cap \Gamma_D = \{0\}$, we impose Dirichlet boundary condition on Γ_D and Neumann boundary condition on Γ_N. The presented formulations allow to have such mixed boundary conditions, by including the Neumann condition into the weak form as a boundary integral, and imposing the Dirichlet condition later, at the discrete level, with the help of (2.4).

In this section, we introduced the necessary mathematical fundamentals to derive the weak form of a PDE problem and to show that under the enlisted conditions the weak formulation has a unique solution. We also presented for the continuous case the embedding of the boundary conditions into the weak form. However, the discrete imposition of the Dirichlet condition, which can have various forms, will be discusset at the end of the following section. In the next section, we discuss the finite element discretization to derive a discrete algebraic system from the bilinear formulation of a PDE problem.

2.2. Finite Element Discretization

Stepping from the continuous to the discrete form of a problem is always crucial in a simulation approach. A continuous solution space, without knowing the analytical solution, implies an infinite dimensional representation. Therefore, for numerical computations, a finite dimensional space is required, to make the problem computable on a computer. In the previous section, we presented the continuous model and the weak formulation of the PDE problem, where the unknown u and test function v are in the infinite dimensional space V:

$$a(u, v) = F(v), \; u \in V, \;\; \forall v \in V, \;\; F \in V^{'}. \tag{2.6}$$

According to Theorem. 2.1.1, the above presented formulation is equivalent to the minimization problem

$$J(u) := \frac{1}{2} a(u,u) - F(u) \longrightarrow \min! \,. \qquad (2.7)$$

The next step is to choose a discretized space for the unknown function $u_h \in U_h \subset V$ and in (2.6) for the test function $v_h \in V_h \subset V$. Such a discrete space S_h can be defined by a basis $\{\psi_1, \psi_2, \ldots, \psi_N\}$, where N is a finite number. Using the spanned discrete space S_h for the unknown (ansatz) and for the test basis, the following relation holds

$$u_h = \sum_{i=1}^{N} y_i \psi_i, \quad v_h \in \{\psi_1, \psi_2, \ldots, \psi_N\}. \qquad (2.8)$$

The vector $\mathbf{y} = [y_1, y_2, \ldots, y_N]$ represents the scaling factor for the basis function and the unknown vector of our discrete problem.

Besides the choice of the discrete space V_h, there is also the choice between equation (2.6) and (2.7). Based on these possibilities we have the following cases:

- **Rayleigh-Ritz-Approach** solves the problem (2.7) where the derivative of $J(u_h)$ in (2.7) with respect to the vector $\mathbf{y}$ is set to zero. The problem formulation for this approach is then to find $u_h \in V_h$ such that

$$(\partial/\partial y_i)\, J\left(\sum_i y_i \psi_i\right) = 0, \quad i = 1, \ldots, N.$$

  In the case of linear form $a(\cdot,\cdot)$, this approach leads to the linear system of equations

$$a(u_h, \psi_i) = F(\psi_i)\,,\ \forall \psi_i \in \{\psi_1, \psi_2, \ldots, \psi_N\}.$$

- **Galerkin-Approach** is the general name, whereas for symmetric bilinear form this is referred to as **Ritz-Galerkin-Approach**. This implies the same discrete space for the unknown (ansatz) and test space $U_h = V_h$, $u_h \in V_h$. The discrete problem formulation for this approach is to find $u_h \in V_h$ such that

$$a(u_h, \psi_i) = F(\psi_i)\,,\ \forall \psi_i \in \{\psi_1, \psi_2, \ldots, \psi_N\}.$$

- **Petrov-Galerkin-Approach** solves also (2.6), but uses different discrete spaces for the test and unknown space. This approach is not wide spread and is used e.g., for singular problems [21].

In this thesis, only the Galerkin-Approach (for symmetric bilinear forms Ritz-Galerkin-Approach) is used, since this is the common way to set up the discrete system of a PDE problem with the finite element method.[7] One obvious technical advantage of this approach is that the same discrete space V_h can be used for test and ansatz functions.

[7] The Rayleigh-Ritz-Approach should give the same discrete system.

At this point, it is important to mention the **orthogonality** property of the finite elements. For the simple case, we consider a symmetric bilinear operator $a(\cdot,\cdot)$ and the linear continuous functional $F(v)$. It holds for the continuous solution of $u \in V$ and the discrete solution $u_h \in V_h$

$$a(u,v) = F(v), \quad \forall v \in V,$$
$$a(u_h,v_h) = F(v_h), \quad \forall v_h \in V_h,$$

where V_h is the discretized V space such that $V_h \subset V$. Subtracting the two relations results in

$$a(u-u_h,v_h) = 0, \quad \forall v_h \in V_h,$$

which says that the error is orthogonal to the current solution space V_h. This also implies that the discrete solution minimizes the error in the discrete space V_h. This property is called in the literature as *Galerkin-orthogonality* and is used in the proof of the following lemma.

Lemma 2.2.1 ***Céa's Lemma***: *Given a bounded and coercive bilinear form $a(\cdot,\cdot)$ with the Hilbert space $(V_h,(\cdot,\cdot))$, then the following relation holds for the discrete solution of the problem $a(u_h,v_h) = L(v_h)$, $\forall v \in V_h$ and the continuous solution u:*

$$\|u-u_h\| \leq \frac{C}{\alpha} \inf_{v_h \in V_h} \left(\|u-v_h\|\right),$$

where C and α are the coefficients defined in Def. 2.2.

Lemma 2.2.1 shows that the accuracy of the of the solution u_h mainly depends on the chosen discrete space V_h, since the error in the solution is less than a constant number multiplied with the error of the best possible solution in the function space V_h. This lemma is essential for the error estimation with the FEM discretization.

Discretized Spaces

We consider the computational domain $\Omega \subset \mathbb{R}^d$, where for most problems $d = 2,3$. The question now is how to choose the discrete functional space V_h, such that it is finite-dimensional (and the vector of unknowns is $\{y_1,\ldots,y_N\}$ from (2.8)). In this section, we highlight the different aspects of the discrete space V_h. The first step is to divide the domain Ω into elementary objects, which we call elements.[8] Each element $E_i \subset \mathbb{R}^d$, $i = 1,\ldots,M$ covers a small portion of the computational domain, such that the sum of the elements results in Ω: $\bigcup_{i=1}^{M} E_i = \Omega$. At the same time, the intersection of these elements $E_i \cap E_j = E^{'}$ is not allowed to be measurable in $\mathbb{R}^d$, hence, $E^{'} \subset \mathbb{R}^j, j < d$. The name of the method finite element also comes from this idea, to have a finite number of elements

[8] A more detailed description of the mesh's cell and element will follow in Chapter 4, where we name the elements with the highest dimension also as cells.

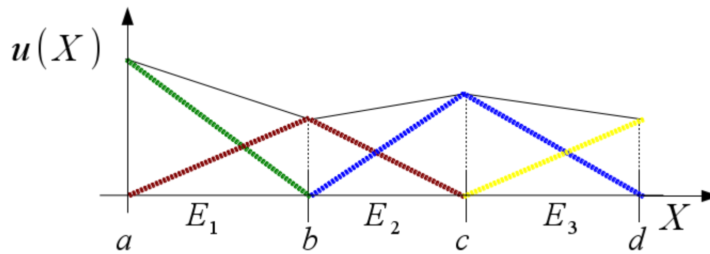

Figure 2.1.: One-dimensional example with the solution function u. The domain Ω is divided in three non-equal elements E_1, E_2 and E_3, and there are four linear basis functions at the points $\{a, b, c, d\}$, marked with different colors.

representing Ω. The elements $E_j, j = 1, \dots M$ form a computational mesh, which is presented in Chapter 5 in a detailed way. There is also the approach to use a mesh-free method, where the elements in this form are not defined, but only points, where the corresponding basis function takes its maximum absolute value.

The elements play an important role in the finite element method. In the following, we consider a one-dimensional example (Fig. 2.1). The one-dimensional Ω is an interval in $\mathbb{R}$ and is divided in 3 elements, which overlap only at the intersection points. The boundary is given by two points a and d. In this case, an element is a line segment, similar to Ω. It is the basic building block to represent a function, e.g., the solution function $u(X)$. On such elements, the FEM uses mostly the basic representation of a function by polynomials.

One of the simplest ways to define a polynomial in 1D is to use a given vector of values $\{(x_1, y_1), (x_2, y_2), \dots\}$. To define a polynomial of order p, one needs $p+1$ points, which the polynomial intersects. This way is a rather intuitive and easy function representation on elements, where we just specify the points x_i and the its function values at these points y_i. The defined polynomial can be formulated as the interpolation polynomial in the Lagrange form, which for the one dimensional case has the form

$$P(x) = \sum_{i=1}^{p+1} y_i L_i(x), \; L_i(x) = \prod_{j=1, j\neq i}^{p+1} \frac{x - x_j}{x_i - x_j},$$

where $\{(x_1, y_1), \dots, (x_{p+1}, y_{p+1})\}$ is given. It is known that this interpolation form is unstable[9] for higher orders. Therefore, this formulation is only used in practical applications up to the sixth order. It is important to note that $L_i(x)$ has a zero value at y_j for all $j \neq i$ and has the value y_i at position x_i. The values y_i are weight factors for the basis function $L_i(x)$. This way, all given nodes $\{x_1, \dots, x_{p+1}\}$ have an associated basis function $L_i(x)$. If we consider the $[0, 1]$ interval, the basis functions for the linear and quadratic case have the forms given in Fig. 2.2. These nodes are also called *local degrees of freedoms* (DoF), since they fix the values at the given coordinates.

[9]due to high oscillations between the nodes x_i and x_{i+1}

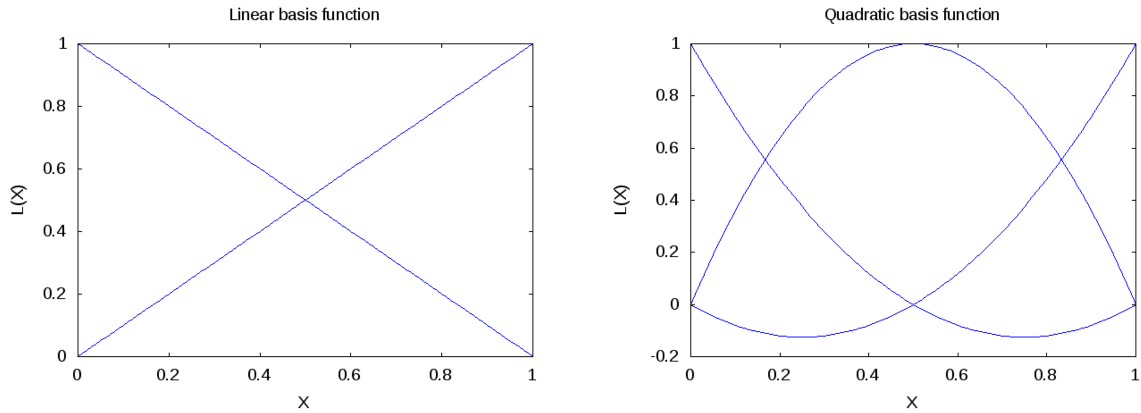

Figure 2.2.: Illustration of linear and quadratic basis functions in 1D. The linear basis (left) with two degrees of freedom, the first located at $x_1 = 0.0$ and the second at $x_2 = 1.0$. The three quadratic basis functions (right) for 1D, where the nodes are located at positions $x_1 = 0.0, x_2 = 0.5$, and $x_3 = 1.0$.

Using the linear polynomial as element basis function, similar to Fig. 2.2, results in piecewise linear function in one dimension as shown in Fig. 2.1. On the interface between elements, it is required to ensure C^0-continuity. For this reason, we use continuous basis function. For the one-dimensional example in Fig. 2.1, we can write the form of the linear basis functions directly using the representation in (2.8), where for the concrete example[10] $N = 4$,

$$L_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if } i > 1 \text{ and } x \in E_{i-1}\,, \\ \dfrac{x_{i+1} - x}{x_{i+1} - x_i} & \text{if } i < 4 \text{ and } x \in E_i\,, \\ 0 & \text{else } (x \notin E_i \text{ and } x \notin E_{i+1})\,, \end{cases}$$

Using this representation and the coefficient vector $\{y_1, \ldots, y_4\}$ the function $u(x)$ can be written as

$$u(x) = \sum_{i=1}^{N} y_i L_i(x)\,.$$

If we consider the element's underlying basis function as a second order polynomial, the equation above still holds. This change would increase the number of basis functions from $N = 4$ to $N = 7$, and the form of $L_i(x)$ would be change to second order as shown in Fig. 2.2. An increase in the order of the basis has the inevitable consequence of an increased number of basis functions per element.

For now, we presented basis functions which are based on the Lagrange interpolation. For this reason, the resulting basis is called *Lagrange basis*, which is widely used for

[10] N is the total number of basis function and is not equal to the number of elements M.

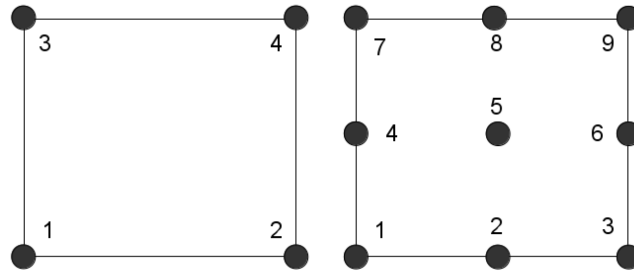

Figure 2.3.: Positions of the nodes in the two-dimensional case for quad elements. The illustration also shows the numbering of the nodes for the bilinear case (left) and for the quadratic case (right).

FE discretization. However, there are other approaches to define a polynomial on these elements in order to represent the solution function. One group of approaches uses the first derivative information at the element interface besides the nodal values. Having the same first derivative on all sides of the element's interface assures C^1-continuity globally on Ω. Such an element is for example the Hermite element, which is based on Hermite polynomials. In this case, some of the basis functions and the coefficients y_k represent the first derivative instead of function values on the boundary of the element.

Different from all basic examples described up to now, hierarchical approaches do not use nodal values, which implies that a basis function is nonzero not only at one node. However, this implies a rather less intuitive representation, which is capable of stable representations even with higher orders. Such a basis is given by the Legendre polynomial, which has been used, e.g., in 2D with up to order 20 [72].

We only use the Lagrange basis in this thesis. For this reason, we take a closer look at the two- and three-dimensional setting. Since Cartesian meshes are in the main focus of this thesis, we present the Lagrange elements only for rectangular cases in the next subsection.

Lagrange Basis Functions in 2D and 3D for Rectangular Elements

For the two-dimensional case, the rectangular element is the quad element, whereas for 3D it is the brick element. These types of elements are the building blocks of the Cartesian mesh, and have rectangular shapes. This type of mesh is presented in Chapter 4, where several advantages of the rectangular elements are shown.

In the following, we introduce the Lagrange basis functions for the two-dimensional case. We start with the linear case and consider the one-dimensional element in Fig. 2.2. For the quad elements we simply take the tensor product of this one-dimensional element, that produces from the two nodes in 1D, four nodes in 2D as shown in Fig. 2.3. This also

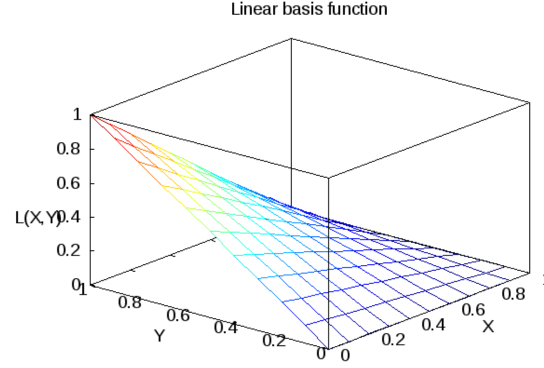

Figure 2.4.: Illustration shows the basis function of the third node (degree of freedom) of the bi-linear quad element.

result from a tensor product of two independent vectors $\{X_{1,1}, X_{1,2}\}$ and $\{X_{2,1}, X_{2,2}\}$. These vectors represent the coordinates of the nodes on the two axes. According to the numbering in Fig. 2.3, these nodes with their associate function values y_i are: $\mathbf{P}_1 = (X_{1,1}, X_{2,1}, y_1)$, $\mathbf{P}_2 = (X_{1,2}, X_{2,1}, y_2)$, $\mathbf{P}_3 = (X_{1,1}, X_{2,2}, y_3)$ and $\mathbf{P}_4 = (X_{1,2}, X_{2,2}, y_4)$. The four bilinear basis functions are defined as

$$L_{2D,1}(\mathbf{x}) = L_{1D,1}(x_1)\, L_{1D,1}(x_2)\,,$$

$$L_{2D,2}(\mathbf{x}) = L_{1D,2}(x_1)\, L_{1D,1}(x_2)\,,$$

$$L_{2D,3}(\mathbf{x}) = L_{1D,1}(x_1)\, L_{1D,2}(x_2)\,,$$

$$L_{2D,4}(\mathbf{x}) = L_{1D,2}(x_1)\, L_{1D,2}(x_2)\,,$$

and $L_{1D,1}$, $L_{1D,2}$ denote the 1D basis functions at the left and right node of the 1D element, respectively. The local coordinates are denoted as $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$. Using the above defined basis functions, the two-dimensional function $u(\mathbf{x})$ can be written as

$$u(\mathbf{x}) = \sum_{i=1}^{4} y_i L_{2D,i}(\mathbf{x})\,.$$

Thanks to the tensor product, the resulting Lagrange basis functions $L_{2D,i}(\mathbf{x})$, similar to the one-dimensional case, have the value zero at all positions $\mathbf{P}_j, i \neq j$, and the value 1.0 at $\mathbf{P}_i$. This is illustrated in Fig. 2.4 for $L_{2D,3}(\mathbf{x})$, where similar to the one-dimensional case, the element is considered on the $[0,1]^2$ domain. This configuration is called the *Reference Element*.

For the quadratic case, we proceed in a similar way. We get the resulting elements basis functions by computing the tensor product of the one-dimensional functions (see Fig. 2.2).

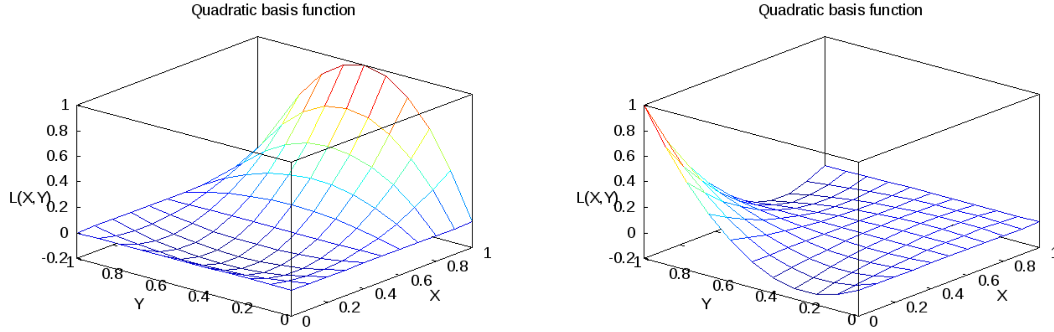

Figure 2.5.: Illustration shows the basis functions of the sixth (left) and the seventh (right) local degree of freedom in the quadratic quad element.

If we consider the full tensor product as shown in Fig. 2.3, there are nine local nodes on the quadratic Lagrange quad element. The general form of the basis functions is

$$L_{2D,k}\left(\mathbf{x}\right) = L_{1D,i}\left(x_1\right) L_{1D,j}\left(x_2\right), \; i,j \in 1,2,3, \; k = 1,\ldots,9.$$

In Fig. 2.5, the quadratic basis functions $L_{2D,6}\left(\mathbf{x}\right)$ and $L_{2D,7}\left(\mathbf{x}\right)$ are illustrated. One can notice that those basis functions, similar to the linear case, have zero values at all neighboring nodes. Even though the element basis functions are quadratic in this case, the function at the element boundary remains only C^0-continuous.

In 3D, the rectangular element, called brick, is in our focus. In this case, we get the basis functions by taking the tensor product between the two- and one-dimensional basis functions.[11] The result can also be seen as a tensor of third order. We get $(p+1)^3$ degrees of freedom per element.[12] These local DoFs for the linear and quadratic case are represented in Fig. 2.6. For higher dimensions, one might observe that the number of local DoFs is growing exponentially[13], with the dimension d. To curve this effect even in lower dimensions (e.g., $d = 2,3$), only a limited part of the tensor product might be used. For example in Fig. 2.3, the local DoF with index 5 might be neglected, still preserving the quadratic representation of the solution. This type of elements is called *Serendipity-element* [21]. In 3D, for quadratic order, one could only use the DoFs on the vertices and the edges of the brick element shown in Fig. 2.6. This would lead to only 20 DoFs compared to the result of the tensor product with 27 nodes, while maintaining the quadratic approximation order of the element. For higher order Legendre elements, only the sparse tensor product is used, such that a drastic increase of element DoFs is avoided by increasing the order in two- and three dimensions [72].

[11] Alternatively $L_{1D,i}\left(x_1\right) \times L_{1D,j}\left(x_2\right) \times L_{1D,j}\left(x_3\right)$, where $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{R}^3$.
[12] p is the approximation order.
[13] This phenomenon is called *curse of dimensionality*.

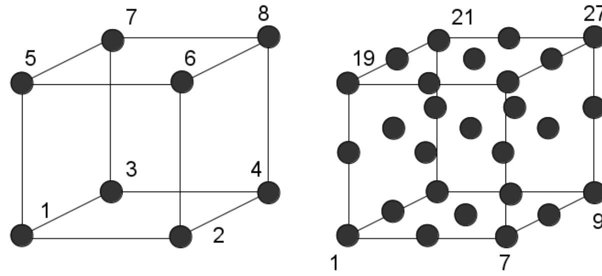

Figure 2.6.: Positions of the nodes in the case of the brick element. The illustration shows the numbering of the nodes in three dimensions for the tri-linear case (left) and for the quadratic case (right).

Outline of the Linear System Assembling

We now defined a regular mesh to represent the computational domain Ω in a discretized form. The building blocks for the discretization are the elements $E_i, i = 1, \ldots, M$. We also defined the function representation on these elements and how C^0-continuity is achieved at the interfaces of the elements. To connect this discretization to the defined weak form of the problem, the next step is to show the setup of the algebraic linear system, which delivers the discrete solution of the problem.

The starting point of the approach is the weak form of the PDE problem in the discrete solution space u_h

$$a\left(u_h, v_h\right) = F\left(v_h\right), \; u_h \in V_h, \;\; \forall v \in V_h.$$

Assuming symmetry for the bilinear form, we use the Ritz-Galerkin approach. To illustrate the method, we consider the Poisson equation, where the linear functional is simply defined by $F\left(v_h\right) = \int_\Omega f v_h\left(x\right) dx$, with a constant value $f \in \mathbb{R}$. With this formulation, the weak form is

$$\int_\Omega \nabla u_h\left(x\right) \nabla v_h\left(x\right) dx = \int_\Omega f v_h\left(x\right) dx. \tag{2.9}$$

Written in a more compact notation, (2.9) can be stated as $\left(\nabla u, \nabla v\right)_\Omega = \langle f, v\rangle_\Omega$. In the following, we use the basis representation defined previously with a finite number N of basis functions

$$u_h\left(x\right) = \sum_{i=1}^{N} y_i \psi_i\left(x\right), \tag{2.10}$$

where the basis functions $\psi_i\left(x\right)$ in our applications are the previously defined Lagrange polynomial $L_i\left(x\right)$. The unknowns, which define our discrete solution are the elements of the vector $\mathbf{y}$. Inserting (2.10) into (2.9) results in

$$\int_\Omega \left(\sum_{j=1}^{N} y_j \nabla L_j\left(x\right)\right) \nabla L_i\left(x\right) dx = \int_\Omega f L_i\left(x\right) dx, \; i = 1, \ldots, N \tag{2.11}$$

Equation (2.11) represents the relations between the elements of the unknown vector $\mathbf{y}$, where the factor relating the i-th and the j-th unknown is given by the integral $\int_\Omega \nabla L_i(x) \nabla L_j(x)\, dx$. Assembling all relations results in a linear system of equations

$$A\mathbf{y} = \mathbf{b},$$

where the matrix is a square symmetric $N \times N$ matrix with elements $A_{i,j} = A_{j,i} = \int_\Omega \nabla L_i(x) \nabla L_j(x)\, dx$, and the right-hand side $\mathbf{b}$ accordingly a column $N \times 1$ vector with elements $b_i = \int_\Omega f L_i(x)\, dx$. However, at this stage, the problem is not solvable, and the matrix A is singular, since the Dirichlet boundary conditions were not integrated yet. Later in this section, we present two approaches to deal with this issue.

Before we bring the mesh into play, we take a closer look at Equation (2.11). This equation does not assume any underlying mesh for the basis functions L_i. This implies that the basis function could be placed in an arbitrary way in Ω. We consider an arbitrary basis function $\psi_i(x)$ that has an associated node[14], where the function has its absolute maximum value. These nodes can be randomly distributed in Ω. This idea is underlying the mesh-free methods, which the finite element method perfectly fits with. One of the disadvantages of such approaches, is the computation of the coefficients $A_{i,j}$. The numerical integration should be done on the intersection of the two basis functions' supports. One other disadvantage is that the resulting matrix can potentially be dense, which makes the solution process of the system more challenging.

Using a mesh and corresponding elements, in our case Lagrange elements, eliminates both disadvantages. Firstly, the mesh structure defines the basis functions that have overlapping support, and, hence, all nonzero entries $A_{i,j}$. Functions that do not have overlapping support by definition have zero contributions $A_{i,j} = 0$. The mesh element also defines the domain for the numerical integration. Instead of integrating on whole Ω, the integration is restricted to a few elements E_i. In the case of the Lagrange basis[15], two basis functions with indices i and j have overlapping support if they share at least one element E_k.

We recall the presented Lagrange elements, where all basis functions belonging to one element, have a common support and the resulting $A_{i,j}$ coefficients are nonzero. This way, the mesh gives a structure for the matrix assembly, where the integration on Ω is transformed into a sum over integrals over the elements with the already mentioned condition $\bigcup_{i=1}^{M} E_i = \Omega$, $E_i \cap E_j = 0$, $\forall i \neq j$:

$$A_{i,j} = \int_\Omega \nabla L_i(x)^T \nabla L_j(x)\, dx = \sum_{E_k \in H_{i,j}} \int_{E_k} \nabla L_i(x)^T \nabla L_j(x)\, dx, \tag{2.12}$$

with $H_{i,j}$ representing the set where $L_i(x)$ and $L_j(x)$ have a common support. In a

[14] Similar to the Lagrange basis $L_i(x)$

[15] This can be generalized for other element types as well.

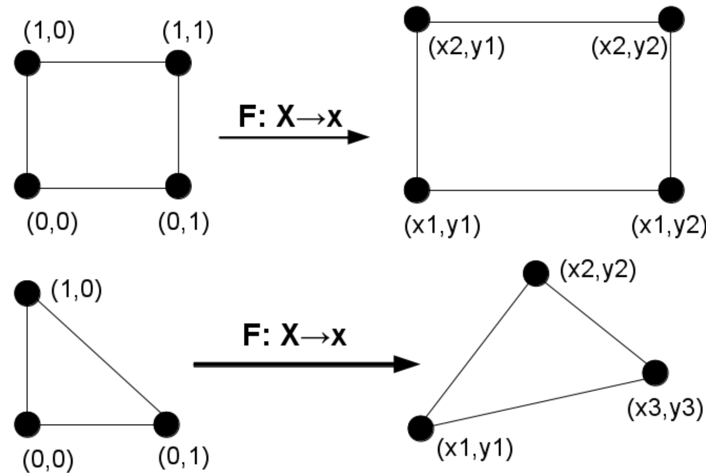

Figure 2.7.: Various affine transformations in two dimensions from the reference element to the real element. This mapping is defined by the $F(\mathbf{X}) : \mathbf{X} \longrightarrow \mathbf{x}$, with the form $F(\mathbf{X}) = B\,\mathbf{X} + \mathbf{d}$, where X is the reference coordinate and x is the real coordinate. Top: regular quad element transformation; bottom: affine triangle transformation.

similar way, the right-hand side integral is reformulated as

$$b_i = \int_\Omega f L_i(x)\, dx = \sum_{E_k \in G_i} \int_{E_k} f L_i(x)\, dx. \tag{2.13}$$

G_i denotes the set of all elements E_k which form the support of the basis function $L_i(x)$. The Poisson equation is a good example to illustrate the practical aspects of the matrix assembly by computing the terms in (2.12) and (2.13). The current formulation implies the computation of the integrals on each E_k. This can be done in a more computationally efficient way, by computing the integral only on the *reference element* [16] and transforming the result to the current E_k element. This transformation is the key for the efficient integral computations in (2.12) and (2.13).

Mappings from the reference element to the mesh's element can have in general various forms. The most general transformation is the isoparametric case, where any continuous mapping can be represented.

Using the notation from Fig. 2.7, we denote the coordinates on the reference element with $\mathbf{X}$, and the coordinates on the element E_i with $\mathbf{x}$.[17] The affine mapping from the reference to this real coordinates $\mathbf{x}$ is represented by $F(\mathbf{X})$, which has the general form $F(\mathbf{X}) = B\,\mathbf{X} + \mathbf{d}$. With this mapping, the integral is transformed from the element E_i to the reference element RE. The reference element has the corresponding basis functions

[16] The element defined on $[0, 1]^d$, introduced in the previous section.
[17] Also called real or physical coordinates.

$L_i(x)$ and $L_j(x)$. We denote these functions with $\phi_i(X)$ and $\phi_j(X)$, respectively. For this affine transformation of the element integrals, the Jacobian of F^{-1} is needed, that is $D_x F^{-1} = B^{-1}$. With these notations, the integration based on the reference cell's integral has the form

$$\int_{E_i} \nabla L_i(x)^T \nabla L_j(x)\, dx = \int_{RE} \left(B^{-T}\nabla\phi_i(X)\right)^T \left(B^{-T}\nabla\phi_j(X)\right) \left|det\, B^{-1}\right| dX. \tag{2.14}$$

Equation (2.14) implies that the integration can be done on the reference element, and only the transformation from the real element E_i to the RE element is needed, which for the affine case is simply a 2×2 or a 3×3 matrix. The right-hand side integral is transformed in a similar way:

$$\int_{E_i} f L_i(x)\, dx = \int_{RE} f\phi_i(X) \left|det\, B^{-1}\right| dX. \tag{2.15}$$

With the help of the (2.12), (2.13), (2.14), and (2.15), the linear system of equations of the discretized Poisson problem

$$A\mathbf{y} = \mathbf{b} \tag{2.16}$$

can be assembled, based only on reference element integrals and on element-wise affine transformations. At this stage, the particular discrete problem is singular due to the missing imposed Dirichlet boundary condition.[18]

Imposing Dirichlet BCs

In the last part of this section, we enlist different approaches to impose Dirichlet boundary conditions at the discrete system level. The condition is written as $u|_\Gamma = g$, which requires that all unknowns y_i located on Γ must have the respective values of g. In the following, we show two methods to impose such conditions discretely.

The first approach sets the coefficients y_i as known values, eliminates the i-th row from the system (2.16), and replaces the i-th value of the right-hand side with $g_{D,i}$. To illustrate this, we consider the i-th row's replacements in the (2.16) system.

$$\begin{pmatrix} A_{1,1} & \cdots & A_{1,i} & \cdots & A_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1.0 & \cdots & 0.0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,i} & \cdots & A_{N,N} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ g_{D,i} \\ \vdots \\ b_N \end{pmatrix}$$

[18] $rank(A) = N - 1$

With this replacement, one forces the unknown y_i algebraically to the value $g_{D,i}$. Further, we denote $\mathbf{g_D}$ the vector with the required Dirichlet values and $\widetilde{\mathbf{y}}$ as the resulting coefficients. The unknowns, which are not impacted, are denoted by $\mathbf{y}^{'}$ and the corresponding right-hand side $\mathbf{b}^{'}$. By replacing all the rows belonging to the index set H_{Γ}, the system is rewritten in a block structured form

$$\begin{pmatrix} A_1 & A_2 \\ 0 & I \end{pmatrix} \begin{pmatrix} \mathbf{y}^{'} \\ \widetilde{\mathbf{y}} \end{pmatrix} = \begin{pmatrix} \mathbf{b}^{'} \\ \mathbf{g_D} \end{pmatrix}.$$

Since the real unknowns are only in the $\mathbf{y}^{'}$ vector, the system simplifies to

$$A_1 \mathbf{y}^{'} = \mathbf{b}^{'} - A_2 \mathbf{g_D},$$

since $\widetilde{\mathbf{y}} = \mathbf{g_D}$. The block matrix A_1 is the decomposition of the matrix formed by the unchanged rows of A. The resulting system is reduced in size, where only the real unknowns $y^{'}$ need to be determined. Unknowns located on Γ are factored on the right-hand side vector.

The second approach is more general and applies not just to nodal basis functions, where a point-wise value can not be enforced. An example is the hierarchical Legendre basis. In these cases, the Dirichlet condition implies

$$\oint_{\Gamma} v(u - g_D) dx = 0 \, \forall v \in V_{h,\Gamma}. \tag{2.17}$$

The discrete function space $V_{h,\Gamma}$ contains the functions which have nonzero values on Γ. We consider only the $\widetilde{\mathbf{y}}$ unknowns, which have a measurable support on Γ. Integration (2.17) results in the discrete system

$$D\widetilde{\mathbf{y}} = \mathbf{g}. \tag{2.18}$$

The matrix D represents the mass matrix resulting from $\oint_{\Gamma} uvdx$ and the right-hand side vector results from $\oint_{\Gamma} g_D v dx$. The system (2.18) has also full rank. Once this system is set up, the next step is to integrate it into (2.16). Similar to the previous approach, we make row replacements, which results in the following block structure

$$\begin{pmatrix} A_1 & A_2 \\ 0 & D \end{pmatrix} \begin{pmatrix} \mathbf{y}^{'} \\ \widetilde{\mathbf{y}} \end{pmatrix} = \begin{pmatrix} \mathbf{b}^{'} \\ \mathbf{g} \end{pmatrix}. \tag{2.19}$$

Since the solution of the vector $\widetilde{\mathbf{y}}$ is decoupled from the other unknowns, it can be solved separately:

$$\begin{aligned} \widetilde{\mathbf{y}} &= D^{-1}\mathbf{g}, \\ A_1 \mathbf{y}^{'} &= \mathbf{b}^{'} - A_2 \widetilde{\mathbf{y}}. \end{aligned} \tag{2.20}$$

However, for technical reasons one might just choose to solve the coupled system (2.19), since it does not imply the refactoring of the original matrix A. On the other hand,

solving (2.20) involves the solution of two systems, each having smaller size than the single system (2.19).

The actual solving of these systems is not in the focus of this thesis. The optimal solver depends on the properties of the matrix, whereas these properties depend not just on the underlying PDE, but also on the chosen element basis function.

Closing Remarks

In this chapter, we introduced the functional analysis basis for the weak form of the PDE problem, and we showed the setup of mesh-based and element's basis function-based discretization. However, further topics remain uncovered in this chapter. One of them is the error indicator based mesh refinement. The derivation of error indicators based on the weak formulation of problems is crucial for optimal mesh refinement. For further details on these topics, we refer to [21, 23, 50].

3. Governing Equations in the Applications

This chapter introduces the governing equations of the various physical systems, which we simulated with our toolbox-approach. We want to emphasize here, that our implementation is limited neither to these equations nor to the scenarios that we set up. The simulated systems demonstrate the capabilities of the created toolbox for immersed boundary approaches. For single-physics applications, we already introduced the Poisson equation in the previous section, which models only the diffusion process. In the following, we introduce the PDEs for viscous flows, namely the Stokes and Navier-Stokes equations. Next, we continue with the elastic body model used in structural mechanics applications. For both models, we discuss the stationary and the transient cases as well as the aspects regarding their spatial discretization. In the last section, we consider as a multi-physics application the fluid-structure interaction (FSI) problem in various configurations. This type of problems requires coupling of the two systems at a given interface. Therefore, it requires interface coupling. The mathematics of this interface coupling is introduced in the last section of this chapter.

3.1. Fluid Model

This section introduces the governing equation of incompressible viscous flows. The flow field is described by a velocity vector field, denoted with $\mathbf{v}$. Besides this quantity, we further characterize the flow with a scalar pressure field p and by a density ρ^f that is assumed to be constant in the incompressible case. We start with the incompressible aspects of the flow by considering the mass conservation equation of an infinite small volume V. The change of the mass in this volume is equal to the in- and outflow through the boundary ∂V:

$$\frac{\partial}{\partial t} \int_V \rho^f dV = - \oint_{\partial V} \rho^f \, \mathbf{n} \cdot \mathbf{v} dc.$$

Next, the boundary integral is transformed with the Gauss theorem into a domain integral,

$$\int_V \left(\frac{\partial \rho^f}{\partial t} + div \left(\rho^f \mathbf{v} \right) \right) dV = 0.$$

Since ρ^f is constant for the incompressible flow, the equation is further simplified to the continuity equation in the strong form:

$$div\,(\mathbf{v}) = \nabla \cdot \mathbf{v} = 0 \;\; in\; V. \tag{3.1}$$

The next governing equation in incompressible flows is the conservation of momentum,

$$\rho^f \frac{d\mathbf{v}}{dt} = \nabla \cdot \sigma_f\,(\mathbf{v}) + \mathbf{f},$$

where $\mathbf{f}$ represents the external forces and $\sigma_f\,(\cdot)$ is the Cauchy stress tensor of the flow. This stress tensor has the definition

$$\sigma_f\,(\mathbf{v}) = \nu^f \left(\nabla\mathbf{v} + \nabla\mathbf{v}^T\right) - pI,$$

where ν^f represents the kinematic viscosity of the fluid. The total difference of $\dfrac{d\mathbf{v}}{dt}$ is further transformed to

$$\frac{d\mathbf{v}}{dt} = \frac{\partial\mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\,\mathbf{v}.$$

With the listed transformations, we get the final form of the momentum equation:

$$\rho^f \frac{\partial\mathbf{v}}{\partial t} + \rho^f\,(\mathbf{v}\cdot\nabla)\,\mathbf{v} = \nu^f\Delta\mathbf{v} - \nabla p + \mathbf{f}. \tag{3.2}$$

Equation (3.2) in combination with the continuity equation (3.1) form the Navier-Stokes equations on a computational domain Ω

$$\rho^f \frac{\partial\mathbf{v}}{\partial t} - \nu^f\Delta\mathbf{v} + \rho^f\,(\mathbf{v}\cdot\nabla)\,\mathbf{v} + \nabla p \;=\; \mathbf{f} \;\; in\;\Omega, \tag{3.3}$$

$$\nabla\cdot\mathbf{v} \;=\; 0 \;\; in\;\Omega. \tag{3.4}$$

The continuity equation (3.4) ensures, that the fluid stays incompressible, whereas the momentum equation (3.3) has several terms. In the stationary case, the time derivative of $\mathbf{v}$ vanishes in Equation (3.3). The next term is the diffusion operator that models the diffusion of the velocity and is proportional to the viscosity ν^f. The third term is the so-called transport or convective term and represents the transport of the velocity field by itself. This is the only non-linear term in the Navier-Stokes equations that needs to be treated in the linearization. The pressure gradient ∇p also contributes to the momentum equation by forcing the flow from higher pressure to lower pressure domains. In the last term, external forces are considered that are summed in $\mathbf{f}$.

For viscous flows, the convective term might become irrelevant, so the Navier-Stokes equations are reduced to the Stokes equations, which only have linear terms

$$\rho^f \frac{\partial\mathbf{v}}{\partial t} - \nu^f\Delta\mathbf{v} + \nabla p \;=\; \mathbf{f} \;\; in\;\Omega, \tag{3.5}$$

$$\nabla\cdot\mathbf{v} \;=\; 0 \;\; in\;\Omega. \tag{3.6}$$

In order to make the problem well defined, additional BCs are required. We consider only Dirichlet boundary conditions for the velocity field on $\partial\Omega$. This implies $\mathbf{v}|_{\partial\Omega} = \mathbf{g}$. This type of boundary condition includes the *no-slip* boundary condition that is imposed in many practical applications. However, this BC only defines the pressure p up to an additive constant in Ω. Therefore, a point-wise fixing of the pressure might be required.

There are different numerical techniques to solve equations (3.3) and (3.4). In this thesis, we only discuss the *coupled approach*, where both fields $\mathbf{v}$ and p are computed simultaneously, hence, these two fields form the unknown vector. In 3D, the resulting system might become large. In order to save computational effort in the transient case, the *decoupled approach* might be considered. In this case, the pressure is computed with the *Pressure Poisson equation* in advance of the velocities. For more details on this approach we refer to [38]. In the case of the *coupled approach*, the absence of the convective term in Equation (3.3) poses a significant computational advantage, since linear solvers can be applied instead of more expensive non-linear solvers.

FEM discretization

In the following, we introduce the finite element discretization of the Navier-Stokes equations, which also holds for the Stokes equations. The first step is to derive the weak form of (3.3) and (3.4). We choose a suited function space for the velocity components and for the pressure. For the components of $\mathbf{v}$, we use the same discrete space. Due to the partial integrations, we require that $\mathbf{v} \in H^1(\Omega)^d$, with $d = 2, 3$ the dimensionality of the problem. A given Sobolev space $H^1(\Omega)$ implies the existence of weak derivatives. The pressure function can be an element of the Lebesgue space $L_0^2(\Omega)$. According to the unknown space, the test space for the velocity is chosen as $\psi \in H^1(\Omega)^d$ and for the pressure $\xi \in L_0^2(\Omega)$. We denote the $d+1$ dimensional test function of the problem as $\phi = (\psi, \xi)$. Equation (3.3) is tested with the velocity's test function, whereas ξ tests Equation (3.4). After multiplying the momentum equation (3.3) with ψ and the continuity equation (3.4) with ξ, the terms $\int_\Omega -\nabla p \, \psi dx$ and $\int_\Omega \nu^f \Delta \mathbf{v} \, \psi dx$ are further integrated by parts. In the resulting equation, the boundary integrals vanish, such that the weak form of the Navier-Stokes equations is written in the compact notation

$$\begin{aligned} \rho^f \left(\frac{\partial \mathbf{v}}{\partial t}, \psi \right)_\Omega + \nu^f (\nabla v, \nabla \psi)_\Omega + \rho^f ((\mathbf{v} \cdot \nabla)\mathbf{v}, \psi)_\Omega & \\ - (p, \nabla \cdot \psi)_\Omega + (\nabla \cdot \mathbf{v}, \xi)_\Omega - (\mathbf{f}, \psi)_\Omega &= 0. \end{aligned} \tag{3.7}$$

Formulation (3.7) does not include the Dirichlet BCs, but those can be included in the classical way, described in Chapter 2[1].

Next, we choose discrete spaces for the velocity $\mathbf{v_h} \in V_h \subset H^1(\Omega)^d$ and for the pressure $p \in Z_h \subset L_0^2(\Omega)$. There is, however, a certain criterion for the V_h and Z_h spaces that is

[1] If they need to be imposed on the facet elements of the cells.

discussed after the time discretization. In this thesis, we employ only Lagrange elements on rectangular Cartesian mesh cells for the Navier-Stokes equations. These elements are denoted as Q_p, where p is the order of the element. One common element for the Navier-Stokes discretization is the Q_2Q_1 element. In 2D, this means, that the velocity has quadratic basis functions, whereas the pressure is represented by a bilinear basis.

Time Discretization

In this thesis, we employ time discretization of order one and two, and as a first step we introduce the operator $a(\mathbf{v}_h, p_h)$ as

$$\rho^f \frac{\partial \mathbf{v}}{\partial t} = a(\mathbf{v}_h, p_h),$$

using Equation (3.7). At time t_n, we discretize the time derivative with time step Δt as

$$\frac{\rho^f \mathbf{v}_h^{n+1} - \rho^f \mathbf{v}_h^n}{\Delta t} = \theta a\left(\mathbf{v}_h^{n+1}, p_h^{n+1}\right) + (1-\theta)\, a\left(\mathbf{v}_h^n, p_h^n\right). \tag{3.8}$$

The value of θ determines the order of the method. In our applications, we use $\theta = 1$ (implicit Euler) and $\theta = 0.5$ (Crank Nicolson). The continuity equation must be fulfilled for all time steps. Therefore, the time discretization must contain an implicit term. Similar to the notation in [35], we transform[2] the equation further to

$$\rho^f \mathbf{v}_h^{n+1} - \Delta t \theta a\left(\mathbf{v}_h^{n+1}, p_h^{n+1}\right) = \rho^f \mathbf{v}_h^n + \Delta t\,(1-\theta)\, a\left(\mathbf{v}_h^n, p_h^n\right),$$

$$\rho^f \frac{\mathbf{v}_h^{n+1}}{\Theta} - a\left(\mathbf{v}_h^{n+1}, p_h^{n+1}\right) = Rhs^n, \tag{3.9}$$

with $\Theta = \theta \Delta t$ and the right-hand side $Rhs^n = \frac{1}{\Theta}\left(\rho^f \mathbf{v}_h^n + \Delta t\,(1-\theta)\, a\left(\mathbf{v}_h^n, p_h^n\right)\right)$.

Stabilization Method

The resulting discrete system from Equation (3.9), including the Dirichlet BCs, is only solvable, if the discrete spaces V_h and Z_h satisfy the *inf-sup* (also called *LBB*) condition that is described in [38, 21]. By using the same type of finite element, the rule of thumb for these spaces, to satisfy the *inf-sup* condition, is to use a higher order basis for $\mathbf{v}$ compared to the pressure field p. The Q_2Q_1 is such an inf-sup stable element.

However, there is a way to circumvent the inf-sup condition by adding a stabilization term to Equation (3.7). Especially in 3D, the Q_2Q_1 element is becoming expensive to use. Hence, there is a practical need to use Q_1Q_1 discretization instead. Since we are

[2]Since we do not use explicit methods ($\theta = 0$) and the time step $\Delta t > 0$ this transformation is valid.

interested only in the stabilization of Q_1Q_1, we choose the pressure stabilized Petrov-Galerkin (PSPG) method that is a consistent stabilization method. For the theoretical background of the stabilization and for other stabilization methods, see [87, 48, 35]. The PSPG method for the Navier-Stokes implies one additional term in Equation (3.7)

$$\tau\left(h\right)\left(\rho^f\frac{\mathbf{v}_h^{n+1}}{\Theta}-\nu^f\Delta\mathbf{v}_h^{n+1}+\rho^f\left(\mathbf{v}_h^{n+1}\cdot\nabla\right)\mathbf{v}_h^{n+1}+\nabla p_h^{n+1}-R^n,\nabla\xi_h\right)_\Omega, \tag{3.10}$$

where R^n denotes the resulting right-hand side in the strong form

$$R^n=1/\Theta\left(\rho^f\mathbf{v}^n+\Delta t\left(1-\theta\right)\left(\nu^f\Delta\mathbf{v}_h^{n+1}-\rho^f\left(\mathbf{v}_h^{n+1}\cdot\nabla\right)\mathbf{v}_h^{n+1}-\nabla p_h^{n+1}\right)\right).$$

The consistent stabilization is realized by multiplying the gradient of the pressure test function $\nabla\xi$ with the residuum of the momentum equation. For the stabilization parameter $\tau\left(h\right)$, in Equation (3.10), we use a simplified form of

$$\tau\left(h\right)=\min\{\Delta t,\frac{h}{2\left\|\mathbf{v_h}\right\|_{L^2}}\},$$

with Δt as the discretized time step and the mesh resolution h that varies in the mesh. For the Q_1Q_1 element, the diffusion term from (3.10) disappears, since the second order derivative of the linear basis is zero.

3.2. Structure Model

To introduce the elastic body model, we consider an infinitely small volume of material V in the stationary case, which is under the influence of a constant traction $\mathbf{t}$. Our goal is to derive the relation between the acting forces and the resulting displacements $\mathbf{u}$ of the body, while the displacement field is constrained by Dirichlet BC. This section of the thesis is based on the first chapters of [93] and on [91].

We start with the resulting state of this volume V that can be described by the second-order stress tensor $\underline{\sigma}_s$, which has the matrix form in 3D

$$\underline{\sigma}_s=\begin{pmatrix}\sigma_{11}&\sigma_{12}&\sigma_{13}\\\sigma_{21}&\sigma_{22}&\sigma_{23}\\\sigma_{31}&\sigma_{32}&\sigma_{33}\end{pmatrix}.$$

The traction force $\mathbf{t}$ is acting on the surface of V denoted as ∂V. Each component of the traction force $t_i, i=1,2,3$, with the three normal directions $\boldsymbol{n}=(n_1,n_2,n_3)^T$ of the normal vector, results in the nine stress components, such that $\underline{\sigma}_s\mathbf{n}=\mathbf{t}$. For instance traction t_2 produces one normal stress σ_{22} and two shear stresses σ_{21} and σ_{23}. The equilibrium state of this infinite small volume implies, that the total torque has to be zero. Therefore, the equation $\underline{\sigma}_s=\underline{\sigma}_s^T$ must hold. Component wise, this means, that

$\sigma_{ij} = \sigma_{ji}, j \neq i$, thus, the stress tensor only has 6 independent components. We rewrite the stress tensor in vector form $\boldsymbol{\sigma}$ for 3D $\boldsymbol{\sigma} = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{13}, \sigma_{23})^T$.

The momentum balance equation in direction x_1 for the volume V with the size $dx_1 \times dx_2 \times dx_3$ says

$$\left(\sigma_{11} + \frac{\partial \sigma_{11}}{\partial x_1} dx_1 - \sigma_{11}\right) dx_2 dx_3 + \left(\sigma_{21} + \frac{\partial \sigma_{21}}{\partial x_2} dx_2 - \sigma_{21}\right) dx_1 dx_3$$

$$+ \left(\sigma_{31} + \frac{\partial \sigma_{31}}{\partial x_3} dx_3 - \sigma_{31}\right) dx_1 dx_2 + t_1 dx_1 dx_2 dx_3 = 0.$$

Each stress, which is acting in this normal direction, has a contribution if t_1 is different from zero. The derivatives of stresses σ_{11}, σ_{21}, and σ_{31} with respect to x_1 are different from zero, since the external force must be balanced by internal forces. The equation further simplified has the following form:

$$\frac{\partial \sigma_{11}}{\partial x_1} + \frac{\partial \sigma_{21}}{\partial x_2} + \frac{\partial \sigma_{31}}{\partial x_1} + t_1 = 0.$$

By proceeding in the same way for the other two directions, the following equation results:

$$div(\underline{\sigma}_s) + \boldsymbol{t} = 0 \;\; or \;\; \nabla \cdot \underline{\sigma}_s + \boldsymbol{t} = 0 \;\; in\, V \tag{3.11}$$

or in matrix form $L^T \boldsymbol{\sigma}_s + \boldsymbol{t} = 0$. The matrix form of L^T is presented in Appendix A.1. Equation (3.11) represents the stationary case, where there is zero acceleration. Hence, no acceleration term is present.

Stresses are then transformed to strains. These strains are the body's normalized deformation representing compressing, stretching, or twisting distortions in the body of the solid. In 3D, it consist of three axial strains and six shear strains, and it is represented as

$$\underline{\varepsilon}_s = \begin{pmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{21} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{31} & \varepsilon_{32} & \varepsilon_{33} \end{pmatrix}.$$

Similar to the stress tensor, the strain tensor is also symmetric and has six independent components in 3D $\boldsymbol{\varepsilon}_s = (\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, \varepsilon_{12}, \varepsilon_{13}, \varepsilon_{23})^T$. The stress-strain relation is defined by a fourth-order tensor, which we denote as $\underline{C}$. This tensor incorporates the material properties. For our case, we consider only super-elastic materials[3]

$$\underline{\sigma}_s = \underline{C} \cdot \underline{\varepsilon}_s. \tag{3.12}$$

The matrix form of $\underline{C}$ is presented in Appendix A.1 by equations (A.3) and (A.4), where the material parameter ν_s and the Poisson ratio E play an important role.

[3] Plastic deformation is not in the focus of our applications.

Strains are now transformed into displacements. The kinematic or strain-displacement equation defines the relation between strains and displacement. This part is by definition non-linear

$$\underline{\varepsilon}_s = \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T + \nabla\mathbf{u}\nabla\mathbf{u}^T\right). \tag{3.13}$$

However, for small displacements it can be linearized, if $\left\|\nabla\mathbf{u}\nabla\mathbf{u}^T\right\| \ll 1.0$,

$$\underline{\varepsilon}_s \approx \frac{1}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right). \tag{3.14}$$

For large displacements, the non-linear equation (3.13) must be used in order to get correct results. We denote this non-linear operator, which maps the displacements to strains as

$$L_n\left(\boldsymbol{u}\right)\boldsymbol{u} = \boldsymbol{\varepsilon}. \tag{3.15}$$

The matrix form of the non-linear operator $L_n\left(\mathbf{u}\right)$ is presented in Appendix A.1.

At this stage, we introduced all the necessary relations from the external traction force vector to the resulting displacement field $\mathbf{u}$. These relations are represented by equations (3.11), (3.12), and (3.13). Instead of considering the infinite small volume V, we extend these equations to the computational domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, and with the boundary $\Gamma_t \subset \mathbb{R}^{d-1}$, where the traction forces $\mathbf{t}$ are acting. By eliminating the stresses and strains from these relations, the resulting equation becomes

$$L\left(C\, L_n\left(\mathbf{u}\right)\mathbf{u}\right) + \mathbf{t} = 0. \tag{3.16}$$

Next, we consider the test and unknown functions in $H^1\left(\Omega\right)^d$, and we state the weak form of the geometric non-linear stationary elastic body equation

$$\int_\Omega (\boldsymbol{du}^T L) C (L_n(\boldsymbol{u})\boldsymbol{u}) dx + \oint_{\Gamma_t} \boldsymbol{du}^T \cdot \boldsymbol{t} dc = 0, \quad \mathbf{u}, \forall \mathbf{du} \in H^1\left(\Omega\right)^d. \tag{3.17}$$

Compared to the Navier-Stokes equations, Equation (3.17) allows compressibility. On the other side, the discretized form does not imply any condition on the discrete solution space (such as the inf-sup condition).

In the transient case, we define the velocity of the structure as $\mathbf{v} = \frac{\partial \mathbf{u}}{\partial t}$, which will be an additional unknown in our equation. The acceleration $\rho_s \frac{\partial \boldsymbol{v}}{\partial t}$ is an additional term to (3.11)

$$\rho_s \frac{\partial \boldsymbol{v}}{\partial t} + \nabla \cdot \underline{\sigma} + \boldsymbol{t} = 0,$$

where ρ_s represents the structure's density. With this term, Equation (3.17) is extended to

$$\begin{aligned}
&\int_\Omega \rho_s \frac{\partial \boldsymbol{v}}{\partial t} \mathbf{du} dx + \int_\Omega (\boldsymbol{du}^T L) C (L_n(\boldsymbol{u})\boldsymbol{u}) dx + \oint_{\Gamma_t} \boldsymbol{du}^T \cdot \boldsymbol{t} dc = 0 \;\; \mathbf{u}, \mathbf{v}, \forall \mathbf{du} \in H^1\left(\Omega\right)^d, \\
&\int_\Omega \mathbf{dv}^T \left(\mathbf{v} - \frac{\partial \mathbf{u}}{\partial t}\right) dx = 0 \;\; \mathbf{u}, \mathbf{v}, \forall \mathbf{dv} \in H^1\left(\Omega\right)^d.
\end{aligned} \tag{3.18}$$

In the weak form Equation (3.18), the continuous test functions $(\mathbf{du}, \mathbf{dv})$ and the unknown functions $(\mathbf{u}, \mathbf{v})$ are replaced with discrete ones, $\mathbf{du}_h, \mathbf{dv}_h, \mathbf{u}_h, \mathbf{v}_h \in V_h \subset H^1(\Omega)^d$. Similar to the stationary case in (3.18), we test the first equation with the displacement's test function, but the second equation is tested with the velocity's test function. The discretization of the time derivative from (3.18) is the subject of the next section.

Time Discretization

The time discretization of Equation (3.17) can be done as follows: We consider the discrete time step Δt at time t_n

$$\rho_s \frac{\partial \mathbf{v}_h}{\partial t} := \rho_s \frac{\boldsymbol{v}_h^{n+1} - \boldsymbol{v}_h^n}{\Delta t} = \theta \mathbf{a}\left(\mathbf{u}_h^{n+1}\right) + (1-\theta)\,\mathbf{a}\left(\mathbf{u}_h^n\right),$$
$$\frac{\partial \mathbf{u}_h}{\partial t} := \frac{\boldsymbol{u}_h^{n+1} - \boldsymbol{u}_h^n}{\Delta t} = \theta \mathbf{v}_h^{n+1} + (1-\theta)\,\mathbf{v}_h^n. \tag{3.19}$$

In Equation (3.19), the operator $\mathbf{a}(\mathbf{u}_h)$ represents the spatial discretization of (3.16). θ defines the order of the method in a similar way as for the fluid equation. For our applications, we use $\theta = 1.0$ that results in the implicit Euler method. This method is known to have a damping effect on the solution.

3.3. Fluid-Structure Interaction

As a multi-physics application, we consider the physical system given by the interaction of an elastic body and a viscous flow. In literature, this problem is called fluid-structure interaction (FSI). The interaction between these separate single-physics systems, the structure and the fluid, is happening through an interface. Therefore, this systems falls into the category of interface coupled systems. On the other side, single-physics systems can also be coupled in their domains that defines the domain or volume coupled systems.

Turning our attention back to the FSI system, we consider a simple example illustrated in Fig. 3.1. The fluid domain is denoted by Ω_f whereas the solid domain is Ω_s. The interface between these two domains, where the coupling takes place, is denoted by Γ. In the fluid domain, we have the fluid velocity field $\mathbf{v}_f$ and the pressure p, whereas in Ω_s the displacements $\mathbf{u}_s$ and velocity field $\mathbf{v}_f$ describe the state of the solid.

The position of the wet wall Γ is defined by the displacement $\mathbf{u}_s$, and in a transient scenario, Γ changes its position. This is the first quantity that needs to be coupled, and implies that the deformed structure is mapped into the flow field. Since the structure equation is mostly computed in the Lagrangian framework and the fluid is in the Eulerian framework, this coupling condition can potentially pose an overhead. This problem is illustrated in Fig. 3.2, where the actual boundary in the Eulerian sense is defined by the

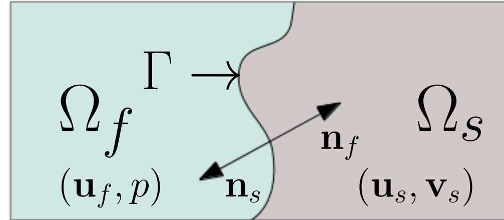

Figure 3.1.: Simple illustration of the fluid-structure interaction with the solid domain Ω_s and the fluid domain Ω_f. The flow is defined by the quantities $(\mathbf{u}_f, p)$, whereas the solid is described by $(\mathbf{u}_s, \mathbf{v}_s)$. The wet wall between these two fields is represented by Γ. On Γ, there are two normal vectors defined $\mathbf{n}_s$ for structure and $\mathbf{n}_s$ for fluid pointing in opposite directions.

Lagrange displacements $\mathbf{u}_s$. We denote $\Gamma = \Gamma_E$ as the Eulerian boundary that is needed for the fluid, whereas the Lagrangian boundary is not changing. Therefore, the following relation holds:

$$\Gamma_E := \{\mathbf{x} + \mathbf{u}_s | \mathbf{x} \in \Gamma_L\}. \tag{3.20}$$

Γ_L denotes the Lagrangian boundary that stays fixed during simulations. In the following, we will only refer to the Eulerian boundary $\Gamma = \Gamma_E$.

To bridge the gap between the two frameworks, several solutions can be used. The most common one is the Arbitrary Lagrange Eulerian (ALE) approach [45] that combines the two frameworks and makes a continuous transition between them in the fluid domain. However, there are also solutions to transform both equations to the same (Eulerian) framework [31]. For further details on these solutions we refer to previous publications [30, 35, 31]. In this thesis, we use the approach, where the two systems are set up in their original framework [25] and the mapping between the two frameworks is done by the interface geometry (see Chapter 8).

The second quantity that is transported from the structure to the flow field is the velocity of the structure at the boundary. Since the wall Γ is wet, in the immediate vicinity of the wall, the fluid must have the velocity of the wall. Therefore, we can write for the velocities of the fluid

$$\mathbf{v}_f = \mathbf{v}_s \quad on\ \Gamma. \tag{3.21}$$

The structure's velocities $\mathbf{v}_s$ serve as Dirichlet values on Γ for the flow equation.

As a reaction to these constraints, the flow is acting with forces on Γ. While the structure sets the Dirichlet velocities on the boundary, the fluid forces are setting the Neumann boundary condition for the structure (see Equation (3.18)). The fluid forces have to be balanced by the reaction forces of the structure. Since forces can be expressed as integrals of stress over the boundary, in the following, we consider the stress vector at the boundary (traction vector $\mathbf{t}$ in Equation (3.18)), and we can write for each point on

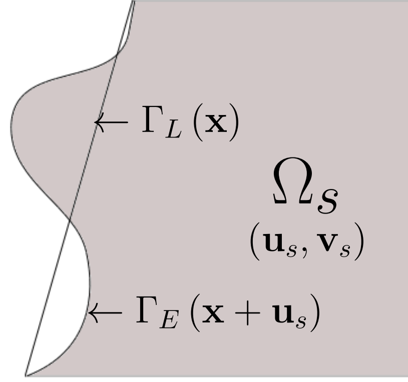

Figure 3.2.: Mapping the Lagrangian boundary Γ_L of the structure to the Eulerian coordinates Γ_E using the displacements $\mathbf{u}_s$, which also define Ω_s.

the interface Γ:

$$\underline{\sigma}_f \mathbf{n}_f = \underline{\sigma}_s \mathbf{n}_s \;\; on \; \Gamma. \tag{3.22}$$

In Equation (3.22), $\underline{\sigma}_f$ and $\underline{\sigma}_s$ are the previously defined stress tensors, and the normal vectors on the opposite sides are $\mathbf{n}_f$ and $\mathbf{n}_s$ (see Fig. 3.1).

At this point, we defined all the necessary conditions of the transient fluid-structure coupling, and they are formulated in equations (3.20), (3.21) and (3.22).

In the stationary case, the velocity of the structure $\mathbf{v}_s$ is by default zero. Therefore, the coupling equations are also simplified to (3.20) and (3.22). On Γ, the fluid's no-slip boundary condition is imposed for the velocity, such that it is consistent with the stationary structure.

4. Cartesian Meshes and Immersed Boundary Methods

Boundary conditions are a crucial part of PDE simulation problems. They usually provide the information that makes the solution unique, such that various numerical methods can be employed for the solutions computation. Incorporating boundary conditions into numerical methods is not straightforward and depends on the overall numerical approach. In Chapter 2, we already highlighted the simple case, when the element's boundary corresponds to the boundary. However, this is not always valid, which can make the imposition of a Dirichlet BC more challenging than the solving itself. In this chapter, we give an overview of various immersed boundary (IB) methods and of Cartesian meshes. The key feature of IB methods is, that they do not require a boundary conforming mesh. Hence, boundary- or geometry-based computationally expensive unstructured mesh generation is avoided. For this reason, the first section of this chapter introduces the Cartesian mesh[1], a memory efficient and adaptive alternative to unstructured meshes, and presents its main advantages in combination with IB methods. The next sections present different approaches for imposing boundary conditions of various PDEs with different discretization techniques on immersed boundaries. In particular, we present Nitsche's method for imposing a boundary condition in a weak sense, which is mainly used in this thesis as IB method.

4.1. Cartesian Meshes

In Chapter 2, we defined the mesh-based discretization of our computational domain $\Omega \subset \mathbb{R}^d$, where for our applications $d = 2, 3$. Such a mesh-based representation of Ω is employed with the FEM in order to obtain a discrete system with a sparse structure in this thesis. In the following, we introduce the Cartesian mesh, which is one focus of this thesis. In this thesis, all the applications are computed on such type of meshes.

[1] also called *regular mesh*.

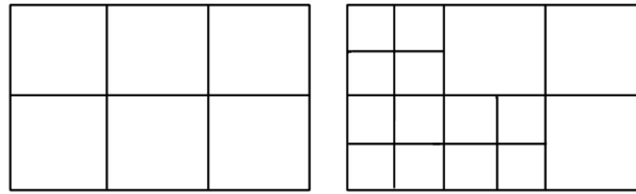

Figure 4.1.: Showing a regular Cartesian mesh (left) and an adaptive Cartesian mesh (right).

4.1.1. Tree-Structured Cartesian Meshes

Structured meshes have the characteristic that their cells' form[2] has a specified global structure. This structure is defined usually by a simple rule. Cartesian meshes are a subgroup of structured meshes with a structure only defined through a Cartesian system $(i\,dx, j\,dy)$ in 2D and $(i\,dx, j\,dy, k\,dz)$ in 3D. Using positive integer indices i and j in 2D with the corresponding upper limits $i < N_x$ and $j < N_y$ results in a mesh with $N = N_x N_y$ cells, similar to the one in Fig. 4.1. The mesh widths dx and dy in 2D can be chosen arbitrarily. Usually, they are chosen such that extensively stretched cells are avoided.[3]

With this simple structure, one can only define regular Cartesian meshes that do not allow adaptivity. Through mesh refinement, local adaptivity can be achieved. This implies usually the replacement of one (parent) cell with several smaller (child) cells. In the case of Cartesian meshes, this refinement is described by the number of divisions per dimension, while keeping the same division number for all dimensions. This way, the refinement usually is either bi- or tri-section. Fig. 4.1 shows a bisection refinement of a regular mesh in 2D.

In the following, we focus on the efficient representation of such adaptive Cartesian meshes. A natural data structure, which fits this purpose, is a *tree*. The tree structure is a special connected and directed acyclic graph, where the connections represent the child-parent relations. Each node in the tree represents a cell. Such a tree must always have a *root node* that does not have any parent cell. Each cell has an associated metric, called the level. It represents the depth of the cell in the tree with respect to the root cell. As illustrated in Fig. 4.2, such a tree represents in a natural way an adaptive Cartesian mesh. Such trees are characterized by the fixed number of children a parent cell has. Accordingly, in 2D with bisection, the result of the tree representation is called a quadtree. In 3D, one obtains an octtree as tree representation with the same refinement strategy. At this point, we want to analyze the storage requirement that a Cartesian mesh induces. Assuming that the computational domain is represented by the root cell,

[2]In the following, we use the term cell instead of element, since the mesh does not have associated basis functions.

[3]where $dx \gg dy$ or $dx \ll dy$.

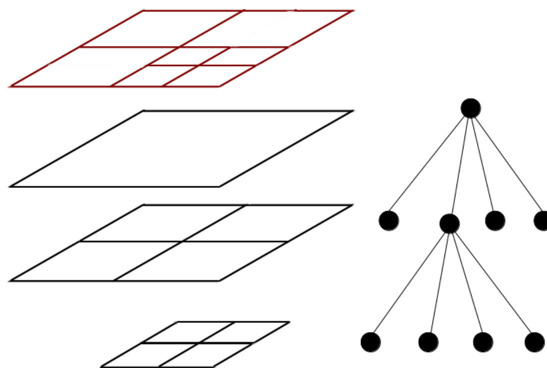

Figure 4.2.: An adaptive Cartesian mesh (left) in 2D represented by a quad-tree (right). Refinement is done by bisection.

the only necessary information at the cell level is whether a cell is refined or not, which can be stored in one bit. This cell-wise refinement information is enough to store the complete structure of the tree. Therefore, it requires minimal storage requirement. It is well known, that these bits can be grouped into a stream of bits, which describes the data structure uniquely. In the case of unstructured meshes, the storage requirement is considerably larger, as one needs to store not just the position of the nodes, but also the connectivity information (e.g., which nodes form the actual cell).

We showed in Chapter 2, that the cells have to fulfill the condition $E_i \cap E_j = 0,\ \forall i \neq j$, for the classical nodal approach.[4] This condition implies, that the cells can not have overlapping measurable domains. For this reason, only the leaf cells form the adaptive computational mesh. To illustrate the leaf view of the mesh we consider the example in Fig. 4.3. The domain Ω is assumed to be rectangular. The initial representation is made by one single cell, and after refinement we end up with two refined cells (black) and seven leaf cell (red). This tree representation of the mesh is employed among others in the *Peano* mesh [92] with trisection refinement. The same concept is used with triangular cells in [5]. The work of Biros et al [86] is also based on this type of tree structure, and the presented algorithms show good scaling with $O\left(10^5\right)$ processors. Using this tree structure [86], the authors implemented an efficient geometrical multigrid solver [78], that was able to solve a system with 8 billion unknowns on 32K processors.

A slightly modified concept is employed in the *p4est* mesh [27]. To illustrate this concept, we consider a further example in Fig. 4.4. Initially, Ω is represented by a regular Cartesian mesh instead of a single cell. After this, some are selected for refinement. This way, a *forest* of trees is created after the refinement, which can be also interpreted as a single tree with $N_x \times N_y$ (in 2D) child cells[5] at the first level. One of the main advantages of this approach is that it saves several refinement steps compared to the

[4]For hierarchical FEM approaches, this condition is usually violated.
[5]N_x and N_y being the initial resolution.

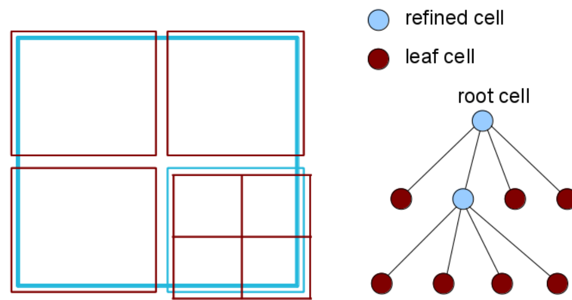

Figure 4.3.: Representation of the domain by a single tree (from Fig. 4.2). The root cell is the only cell at the first level. The leaf cells marked with red are slightly displaced to show the hierarchical structure. The leaf view of the tree represents the adaptive Cartesian mesh.

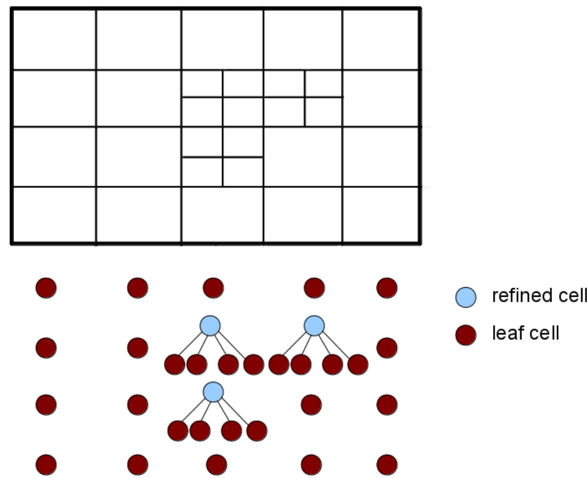

Figure 4.4.: The initial domain represented by a 5×4 regular mesh. Three of the coarse cells were refined (top) resulting in a forest of trees (bottom).

single root cell approach, since most applications require a coarse regular mesh in order to start the refinement process. In addition, this approach has to manage the forest of trees, which proved to be efficient in the massively parallel case [27]. In Sundance, we use this approach for the parallel adaptive Cartesian mesh implementations as show in Chapter 6.

For Cartesian meshes, one can expect considerably shorter traversal and setup times compared to the unstructured mesh approach, where the meshing algorithm poses a significant overhead. Fast multilevel solvers can be applied at hierarchical mesh structures.[6] In this thesis, we do not enable the usage of geometrical multigrid with Cartesian

[6] Algebraic multigrid solvers and preconditioners do not require hierarchical mesh structure, they build one for themselves.

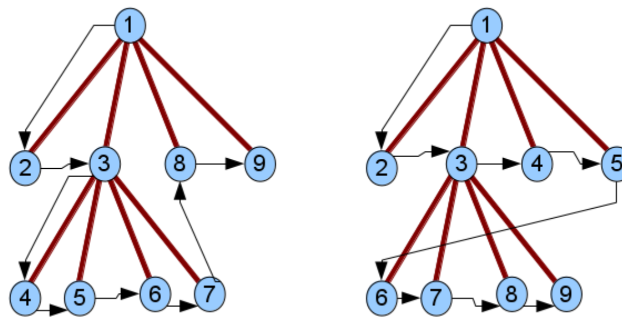

Figure 4.5.: Left depth-first (preorder) traversal, right a breadth-first traversal of the same tree.

meshes, since we restrict ourselves to the FEM-toolbox, and this advantage of Cartesian meshes, in our case, remains unexploited.

4.1.2. Cartesian Mesh Traversal and Domain Decomposition

In this part, we present a particular feature of Cartesian meshes, the traversal along space-filling curves, and its application in decomposing the mesh for parallel computations.

Once a tree-structured Cartesian mesh is defined, the next step is to find a deterministic manner to traverse the tree. The traversal of the mesh is required not just in the solving process of the PDE problem, but also at a later visualization or solution evaluation stage. Especially in a matrix-free solver context, the mesh traversal has a special role. In this case, the total runtime is directly dependent on the efficiency of the traversal algorithm.

In parallel applications, the mesh traversal needs to be done in parallel in order to avoid computational bottlenecks. The parallel traversal along the distribution of cells among processors in a balanced manner is a major task that can be supported by a suitable traversal algorithm, as we show in the following.

In the case of a Cartesian mesh, the sequential mesh traversal is equivalent to a traversal of the underlying tree, which can have different forms. The two main groups of methods are the *depth-first* and the *breadth-first* traversals. The breadth-first traversal visits first all the cells on the current level before it traverses cells at higher levels, whereas the depth-first search visits all children of a node before moving to the neighbor node with the same level. Fig. 4.6 illustrates these two traversal algorithms on the same tree example. Depending on the application one might choose different traversal algorithms, but for our FEM applications, the most convenient one is the depth-first approach as it provides a more or less space-continuous sequence of cells: The refined cells of a coarser cell are traversed before neighbors, for this reason, the traversal has a high spatial locality.

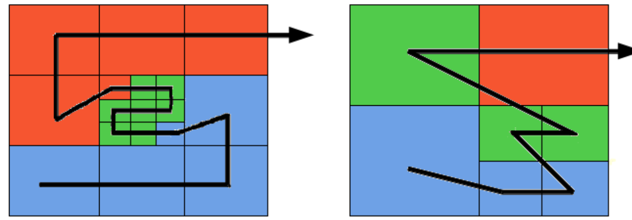

Figure 4.6.: Peano space-filling curve (left) traversal of an adaptive mesh, and the Z-curve (right) traversing a simple refined Cartesian mesh. The three colors illustrate the decomposition of the mesh into subdomains. It also shows that the globally discontinuous curves, such as the Z-curve (right), can potentially produce disconnected subdomains.

Iterates of space-filling curves represent a line which connects all cells in the mesh, and it also represents a mesh traversal technique. Fig. 4.6 shows two types of space-filling curves, the Peano curve for trisection refinement, and the Z-curve for bisection, both in 2D. In this concrete example, only the leaf cells were marked in the traversal, which corresponds to the nodal element criterion that two cells are not allowed to have overlapping domains. The resulting curve is basically a sequence that contains all the cells of the Cartesian mesh.

In the parallel case, it is necessary to decompose the mesh into equal domains, such that each processor has its own subdomain of the mesh. The resulting domains mainly define the quality of parallelization of further computations. Therefore, it is required to distribute the work in a load-balanced manner and at the same time, in order to minimize communication among the processors, all sub domains should have minimal interfaces with other domains. All the enlisted features can be accomplished with the space-filling curve based partitioning. Given the sequence of cells, generated by a space-filling curve, a partitioning of the mesh can be achieved by dividing this sequence into equal pieces. In Fig. 4.6, this is illustrated for a simple partitioning into three subdomains. The Peano curve always generates connected sub domains, whereas the Z-curve can potentially produce disconnected subdomains.

An alternative to the Z-curve, in the case of bisection, is the Hilbert curve, which is globally continuous and, similar to the Peano curve, it produces connected subdomains. In Chapter 6, we discuss in more detail the implementational aspects and implications of a space-filling curve based traversal.

4.1.3. Geometry and Boundary Representation

In this thesis, we only consider undeformed Cartesian meshes (up to different h_x and h_y). Isoparametric mesh transformations [21] would enable a more accurate representation of

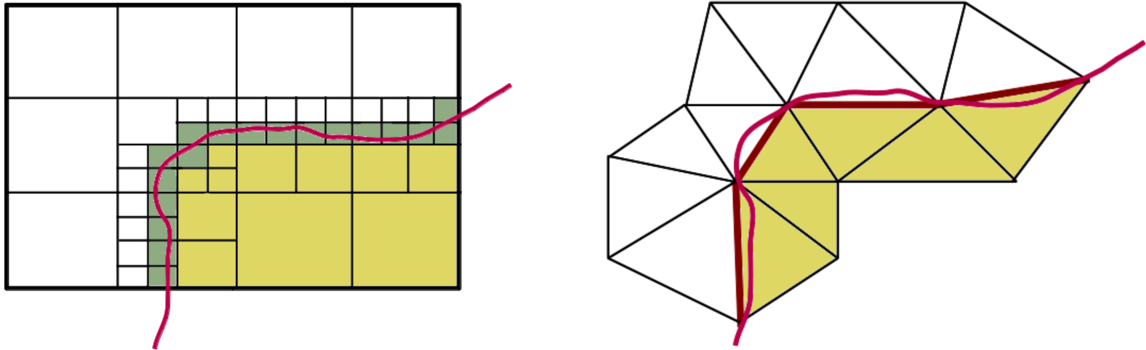

Figure 4.7.: Representation of complex geometries. On Cartesian mesh (left) the geometries cuts the green cells. The edges of these cells could be used as an approximation of the boundary. In the case of unstructured mesh (right) with triangles and even with smaller number of cells, the geometry can be represented more accurately (the marked edges of triangles represent the boundary). The white cells represent the computational domain whereas the yellow ones the domain outside Ω.

the geometry by parameterizing the cell's facets such that it fits the given boundary. This way, higher order (e.g., 2, 3) representation of the boundary is possible. However, such a representation is computationally costly and not feasible for arbitrary large geometry changes in the moving geometry case, where remeshing is required. Therefore, we focus on undeformed meshes in an Eulerian setting.

In the case of unstructured meshes, the nodes of the cells are usually chosen such, that the cell's facets coincide, at least in linear approximation, with the boundary (see Fig. 4.7). This way, one cell can be either in or out of the computational domain. In the case of Cartesian meshes, there is a third group of cells which are cut by the geometry. The geometry could be represented by the facets of these cells, which obviously would lead to an only $O(h)$ geometry approximation, where h is the mesh width on the boundary. This effect is shown in Fig. 4.7, where a complex boundary (marked with red) is intersecting a refined Cartesian and a simplex unstructured mesh in 2D. The unstructured mesh represents the boundary by line segments, which correspond to the cells boundaries, leading to an $O(h^2)$ approximation of the boundary. The Cartesian mesh only has rectangular cells, and correspondingly rectangular line segments. These lines are unsuited for such complex geometries. For this reason, one of the topics of this thesis is to improve this property of the Cartesian meshes. In the next section, we enlist some of the methods for better geometry representation, where the boundary condition can be enforced with more than first order accuracy.

During refinement, level differences between neighboring cells occur. The maximal difference is called the irregularity of the mesh. Fig. 4.7 (left) shows a **2-irregular** mesh, since the highest level difference is two. As the irregularity has severe impact on numer-

ical and implementational efficiency, we limit our meshes to 1-irregular meshes. Thus, only the difference of one refinement level has to be bridged. Such 1-irregular meshes with bisection refinement are also called meshes preserving the **2:1 mesh balance** [6]. In Chapter 6, that deals with the *hanging node issue*, we come back to this criterion.

4.2. Immersed Boundary Methods

In the previous section, we presented the main advantages and the disadvantages of adaptive Cartesian meshes. The main advantages are the low memory requirement, simple data structure even for the adaptive case, and space-filling curve based traversal and domain decomposition. The main disadvantage is the poor capability to represent accurately complex geometry boundaries, on which we imposed Dirichlet boundary conditions. As shown in Fig. 4.7, even with local refinement this disadvantage can not be completely compensated. Since the representation is based only on rectangular cells, the accuracy is only $O\left(h\right)$.

To eliminate this drawback of the Cartesian meshes, we use the *immersed boundary methods* (IB methods). The main idea is to embed the complex geometry into a larger rectangular Ω_O domain, such that $\Omega = \Omega_O - \Omega_F$, with $\Omega \cap \Omega_F = 0$, where Ω_F is called the fictitious domain. In spite of using a Cartesian mesh to discretize Ω_O, the computations should be done on Ω. Cells of the Cartesian mesh that are entirely in or outside Ω are simple to handle. The challenge at this point remains how to handle the cells that contain boundaries, such that a given BC on $\partial\Omega$ is imposed, while solving an equation on the entire Ω_O domain. Fig. 4.8 illustrates an example with complex immersed boundaries.

4.2.1. Overview

With immersed boundaries, there is a need for geometry description, not just to determine the cells which are intersected by it, but also to handle these cells, depending on the chosen IB method. This geometry description can also be used for mesh refinement near the boundary. Such a description is not necessary for unstructured meshes, since the geometry is implicitly described by the facets of the boundary cells (see Fig. 4.7). The implementational aspects of the geometry representation for our concrete case are discussed in Chapter 7. A further advantage of the immersed boundary is visible for moving boundaries, where in the case of unstructured meshes mesh transformations and for topology changes even a costly remeshing are required. All these additional overheads are not present with immersed boundaries, where the mesh is left unchanged, and only the geometry is moved.[7] However, this might have the implication that the cells need to be regrouped, such that cells in Ω_F might become part of Ω and vice versa.

[7]Assuming that we work in the Eulerian ansatz.

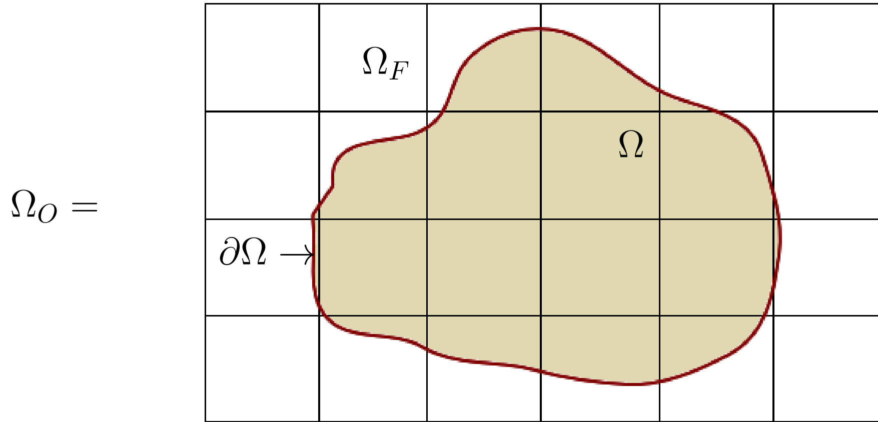

Figure 4.8.: Example for an immersed boundary scenario. Ω is the computational domain which is embedded into a rectangular domain Ω_O that can be discretized with Cartesian meshes. The fictitious domain Ω_F is defined as $\Omega_F = \Omega_O - \Omega$ and $\Omega_O = \Omega \cup \Omega_F$.

There have been numerous publications on methods which can be categorized as IB methods. Depending on which level they impose the BC, we divide them in two main groups: (1) *continuous* or *weakly imposed* methods and (2) *discrete* methods. In the following, we present a general overview of these IB methods before we turn our attention to the specific continuous methods that are employed in our FEM-based PDE toolbox.

Historically, the first (continuous) IB method was introduced in [74] and was used for cardiac mechanics and the coupled blood flow. The elastic wall is represented as a spring which is acting with a given force on the fluid. The force is modeled by a source term in the flow equation. However, this variant of the elastic wall model is not suited for rigid body representation, since a stiff wall response generates usually a stiff system to solve [56].

A more general continuous method, also applicable for rigid obstacles, is the *penalty method*. It adds a term to the formulation that “forces” the solution to the values of g. Methods that also fall into this category have been implemented for advection-diffusion, incompressible flow, and turbulence models. In the turbulent case, additional wall shear-stress boundary terms are included as Neumann boundary conditions in the variational form of the equation [14], enabling the easy use of IB methods. A similar approach is used in [44], where a Poisson equation for the wall-distance is solved with immersed boundaries. This distance is later used in the wall function of the $\kappa - \varepsilon$ model. One of the methods developed for viscous flows is presented in [13], where additional penalty and consistency terms enforce the prescribed boundary velocities. In this category, we also have to mention the Stokes/Navier-Stokes Brinkman equation [24] that models the porous medium by adding one addition permeability force term to the impulse equation, and can model fluid obstacle geometries with the help of varying permeability (see Chapter 9 for more details). Penalty-like methods are presented in

more details in Subsection 4.2.2.

The next continuous approach is the Lagrange multiplier method that considers the boundary condition as constraint for the variational formulation of the problem. This involves the usage of extra degrees of freedom which are the multipliers. These degrees of freedom can be either defined on the interface [35] or in the whole Ω_O domain [42]. The concrete mathematical formulation of this approach is stated in Subsection 4.2.4. The Nitsche-type methods can be seen as a special case of the Lagrange multiplier approach [84]. First introduced in [69], the idea is to define and minimize the energy functional of the variational problem formulation including boundary conditions as constraints. This gives rise to a consistent method to impose the boundary conditions continuously without any additional degrees of freedom. Nitsche's method for the Navier-Stokes equations is one of the major contributions of this thesis, and the theoretical foundation of this method is shown in Subsection 4.2.5.

The extended finite element method (XFEM) can be categorized also as continuous enforcement methods, although it uses additional enriched basis functions for the cells that are intersected by the boundary. These enrichments are defined in such a way that discontinuities at the boundary can be captured. A first version of the method was used for crack tracking in elastic structures [17, 68], and was originally called generalized finite element methods GFEM [85]. This technique facilitates in combination of Lagrange multipliers an efficient implementation of IB in fluid mechanics [35] and fluid-structure interaction with moving boundaries.

Level-set methods use an explicit description of the boundary by a function ϕ. Assuming that this function is continuous, the boundary is defined by the points $\mathbf{x}$ for which $\phi(\mathbf{x}) = 0$. This function ϕ is subject to changes that are usually described by a convection dominated PDE (e.g., $\frac{\partial \phi}{\partial t} + \mathbf{v} \cdot \nabla \phi = 0$, with a velocity vector $\mathbf{v}$). Since this function is changing, the position of the interface $\partial\Omega = \{\mathbf{x} \in \mathbb{R}^d \mid \phi(\mathbf{x}) = 0\}$ is also changing. By extending the interface problem in this extra dimension, the method gains real strength to deal with general interface problems and even with topology changes. This idea was introduced in [70] and a method similar to the level-set approach is also used for fluid-structure interaction with IBs in [31].

The finite cell method (FCM) is a special method to compute the deformations of an elastic structure with complex shapes [72]. Key idea of the approach is to use an elasticity matrix $C(\mathbf{x})$ that depends on the position $\mathbf{x} \in \mathbb{R}^d$. This approach is discussed in more details in Subsection 4.2.3.

Besides *continuous* enforcement methods, there is also a *discrete* way to enforce IB conditions. One way is to estimate a priori an algebraic forcing term that is added to the right-hand side, such that the corresponding DoF has the required value [89]. A more direct and general way is the ghost-cell approach, where the values in the fictitious domain Ω_F are set such that the interpolant on $\partial\Omega$ takes exactly the values of g. This is a

common way to enforce IB conditions in the context of finite differences and is employed for flow simulation with complex geometries [52]. A similar discrete approach is also presented in [20] for IBs. The next type of discrete approach is the cut-cell approach for finite volume discretizations. The main idea of this method is to take into consideration only the Ω part of the cell in the computations [66]. For more insights and comparison of these methods, we refer to the review paper [67].

The scope of the rest of this chapter is a more detailed presentation of various immersed boundary methods that can potentially be used in a FEM-based PDE toolbox and Cartesian mesh context. For this reason, we only list the methods that enforce weakly the boundary condition on such immersed boundaries. This implies that the methods should be stated in a weak form, similar to the formulation in Chapter 2. In contrast to the general and discrete methods to impose the Dirichlet boundary conditions, presented in Chapter 2, some of the following methods are PDE specific or valid only for one type of problems. In Chapter 7, Chapter 8, and Chapter 9, two methods are employed in concrete applications, where we demonstrate the capabilities of these immersed boundary methods in combination with Cartesian meshes.

4.2.2. Penalty Method

The first group of methods to be introduced is a simple and general one, which we call simply as penalty methods. We consider the solution function $u \in \mathbb{R}^d$ on the computational domain $\Omega \subset \mathbb{R}^d$ and the boundary condition $u|_{\partial\Omega} = g$, with $\partial\Omega \subset \mathbb{R}^{d-1}$. The main idea of this method is to add a penalty term to the weak formulation of the problem which *penalizes* values of u on $\partial\Omega$ that deviate from g.

In general, the following penalty term is added to the original weak form of the equation

$$\int_V \alpha \left(u - g\right) v \, dx, \forall v, \tag{4.1}$$

where α is the penalty coefficien, and V is either Ω_F or $\partial\Omega$. There are two main groups of penalty methods, depending on the domain V of penalization term. The first one, is the *volume penalty method*, where a penalty force is acting on Ω_F. The Brinkman-type [24] equations for flow simulation fall into this category, where a homogeneous boundary condition of the velocity vector $\mathbf{u}_{\partial\Omega} = \mathbf{g}$ is imposed in a general way on a complex geometry. For a concrete example, we consider a flow scenario and the setting in Fig. 4.8, with Ω as the fluid domain and Ω_F as the fictitious domain denoting the solid domain. In this case, we assume that g is defined on all Ω_O. [2, 51] and [76] use this approach, where the momentum equations has one additional force term of the following form

$$\int_{\Omega_O} \frac{\mu}{\mathbf{k}\left(\mathbf{x}\right)} \left(\mathbf{u} - \mathbf{g}\right) \mathbf{v} \, dx, \tag{4.2}$$

where μ is the viscosity of the fluid, $\mathbf{u} \in H^1(\Omega_O)^d$ is the velocity vector and $\mathbf{v} \in H^1(\Omega_O)^d$ is the test function. $\mathbf{k}(\mathbf{x})$ represents the coordinate dependent permeability that in the case of the solid is close to zero ($k_i \ll 1$). The permeability has high values ($k_i \gg 1$) in the fluid domain, hence, term (4.2) vanishes in Ω

$$k_i(\mathbf{x}) = \begin{cases} k_F & , k_F \gg 1 \ \mathbf{x} \in \Omega \\ k_S & , k_S \ll 1 \ \mathbf{x} \in \Omega_F. \end{cases} \tag{4.3}$$

Depending on the domain, where term (4.1) is defined, [76] presents two variants. *Exterior penalization* defines term (4.1) on Ω_F, which is the case with the permeability coefficient defined in (4.3) and is the presented *volume penalty* approach for fluids. The *spread interface penalization* uses the penalty term (4.1) only on the approximation of $\partial\Omega$.

The *spread interface penalization* represents the next group of penalty methods, namely the *interface penalty* methods, where $V = \partial\Omega$. This approximation is a simplified (e.g., a piecewise linear) representation of the boundary, and (4.1) is transformed to a boundary integral

$$\oint_{\partial\Omega} \alpha(\mathbf{u} - \mathbf{g})\,\mathbf{v}\, dc. \tag{4.4}$$

To illustrate this concept, we consider an example enforcing the boundary condition with a penalty term. The concrete PDE is the Poisson equation $-\Delta u = f$ in Ω, $u = g$ on $\partial\Omega$. The weak formulation of the problem including the penalty term is

$$\int_{\Omega} \nabla u \nabla v - fv\, dx + \oint_{\partial\Omega} -v(\nabla u \cdot \mathbf{n}) + \frac{\gamma}{h} v(u-g)\ dc = 0. \tag{4.5}$$

$u \in V_h(\Omega_O)$ and $v \in V_h(\Omega_O)$ are the unknown and test functions and $\mathbf{n}$ is the normal vector pointing outwards of the domain Ω. h is the mesh width on $\partial\Omega$. Penalty terms are usually scaled with the inverse of the mesh width $\alpha = \frac{\gamma}{h}$, such that the method is convergent for $h \to 0$ while $\alpha \to \infty$. The boundary integral term $v(\nabla u \cdot \mathbf{n})$ is the result of the partial integration, which does not vanish in the IB case.[8]

Equation (4.5) is employed for the example geometry in Fig. 4.9. The fictitious domain is inside the circle, which is marked with white color. On this circle, we impose a constant BC $g = 1.0$. The error norm measured on the boundary is only $\|u-g\|_{L^2(\partial\Omega)} = 9.7e-3$, which shows the high accuracy of the weak enforcement of BC on even coarse Cartesian meshes that are not conforming with the domain boundary.

A similar approach is described in [13] for flow problems. The authors apply this IB approach for advection-diffusion and for incompressible flow scenarios. The key idea is to extend the pure penalization concept, by adding additional penalization and consistency terms to the weak formulation of the advection-diffusion equation, $a\nabla u - \kappa\nabla(\nabla u) = f$,

[8] u and v do not have compact support on Ω, and the BC is enforced weakly.

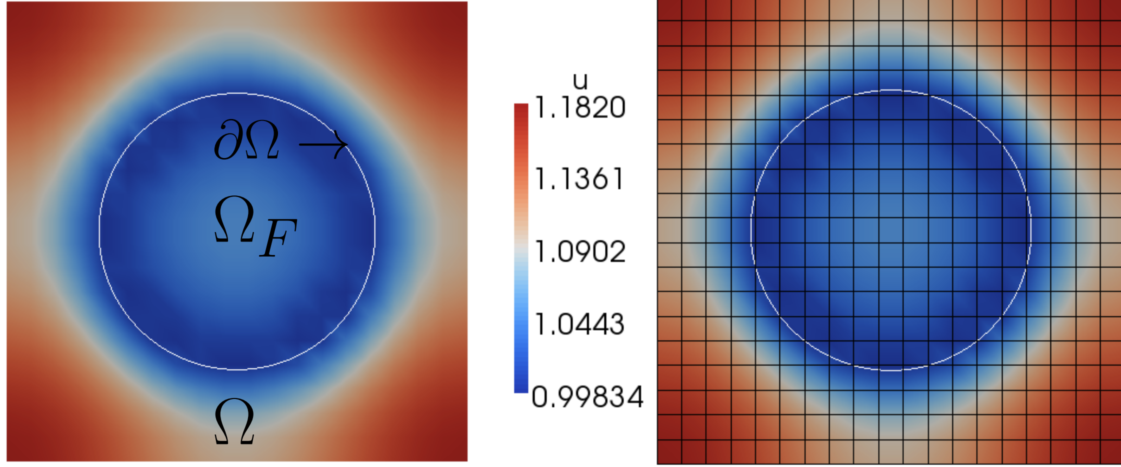

Figure 4.9.: Solution of the Poisson equation in the weak form (4.5), with $\gamma = 6.0$. The BC is constant with $g = 1.0$ on $\partial\Omega$, that is a circle, located in the middle of Ω with a radius of 0.3. The source term is also constant with $f = 1.0$. The fictitious domain Ω_F is represented by the domain inside the circle. On the boundary of the rectangle, we impose Neumann zero boundary conditions. The resulting values are shown on the left, whereas the right plot shows also the underlying mesh. Computations were made with the Sundance toolbox [61] that is presented later in Chapter 5.

with the convective coefficient a and κ as the diffusion coefficient. The IB conditions can be imposed weakly in the following way (in a simplified form): for $\mathbf{u} \in V_h(\Omega_O)$ and $\forall \mathbf{v} \in V_h(\Omega_O)$

$$\int_{\Omega} \left(-\nabla \mathbf{v} \left(a\mathbf{u} - \kappa \nabla \mathbf{u}\right) - \mathbf{f}\mathbf{v}\right) dx + \oint_{\partial\Omega} \mathbf{v} \left(-\kappa \nabla \mathbf{u} \cdot \mathbf{n} + a\mathbf{n} \cdot \mathbf{u}\right) dc$$

$$+ \oint_{\partial\Omega} -\gamma\kappa \nabla \mathbf{v} \cdot \mathbf{n} \left(\mathbf{u} - g\right) + \frac{C\,|\kappa|}{h} \mathbf{v} \left(\mathbf{u} - \mathbf{g}\right) dc = 0. \quad (4.6)$$

The domain integral represents the weak form of the advection-diffusion equation. The third integral term is called the consistency term in [13], and it arises from the partial integration of the strong from. These terms, in contrast to the case of boundary conforming meshes, do not vanish, since v does not have compact support on Ω. The last boundary integral enforces the Dirichlet boundary condition[9] with $\gamma = 1$ or -1, C being a penalty coefficient, and h the mesh width on the boundary. The authors of [13] claim that with $\gamma = 1$ or -1, the resulting method is consistent and they apply the same principle for the Navier-Stokes equations. We show here the improved method presented

[9]We stated it in a simplified form, where we consider only one type of boundary (no separate in- and outflow).

in [14]. We denote the additional pressure scalar field with p, and the associated test function with q. For $\mathbf{u}, p \in V_h$ and $\forall \mathbf{v}, q \in V_h$,

$$\int_\Omega \nabla \mathbf{v} \kappa \nabla \mathbf{u} + \mathbf{v}\left(\mathbf{u}\cdot\nabla\right)\mathbf{u} - \mathbf{f}\mathbf{v} - \nabla \mathbf{v}\, p + q \nabla \mathbf{u}\; dx$$

$$+ \oint_{\partial\Omega} \mathbf{v}\left(-2\kappa \nabla \mathbf{u}\cdot\mathbf{n}\right)\; dc + \oint_{\partial\Omega} -\gamma 2\kappa \nabla \mathbf{v}\cdot\mathbf{n}\left(\mathbf{u}-\mathbf{g}\right) + \frac{C\left|\kappa\right|}{h}\mathbf{v}\left(\mathbf{u}-\mathbf{g}\right)\; dc = 0. \quad (4.7)$$

Equation (4.7) has the same structure as (4.6) with the additional consistency and penalty terms. This formulation of the problem is used in [14] for turbulent flow scenario simulation, where due to the IB method a lower wall refinement is required.

4.2.3. Finite Cell Method

The Finite Cell Method (FCM) is a general method to impose boundary conditions for the elastic structure equation introduced in Chapter 3. This method was proposed in [72] for higher-order Legendre basis functions and was extended later for B-spline basis functions in [81, 82]. The main idea of the FCM is to define a coordinate dependent elasticity matrix. This dependency is induced by a coordinate dependent coefficient α

$$\alpha = \begin{cases} 1.0 & , \mathbf{x} \in \Omega \\ 0.0 & , \mathbf{x} \in \Omega_F\ . \end{cases} \quad (4.8)$$

Using this coefficient in the linear elasticity equation, it results in the following weak formulation of the problem. Similar to the definition in Chapter 3, $\mathbf{u}$ denotes the displacements in $d = 2$ or 3, $\mathbf{v}$ is the corresponding test function, and the weak form of the FCM has the form:

$$\int_\Omega \left(L\,\mathbf{v}\right)^T \alpha C\left(L\,\mathbf{u}\right) + \alpha \mathbf{f}\mathbf{v}\; dx + \oint_{\partial\Omega} \mathbf{t}^T \mathbf{v} dc = 0. \quad (4.9)$$

$\mathbf{f}$ represents the volume forces and $\mathbf{t}$ is the traction force on the surface of the elastic body, and L is the differential operator from Chapter 3. The weak form (4.9) is solved on an adaptive or even regular Cartesian mesh.

To illustrate this concept, we consider the perforated plate benchmark scenario that we compute with the FCM method. The computational domain is illustrated in Fig. 4.10. The lower part of the plate is fixed. At the top boundary, a force of 100N is acting in upwards direction. In this case, the goal is to compute a correct displacement field. The stress fields are also of interest, but they might be unstable with Legendre basis functions in the nonlinear case. Therefore, an improved version of the FCM is proposed in [81], which uses hierarchical enriched B-spline basis.[10] Since we are in the linear case, this improvement is not necessary for our example. We use the classical FCM with a coarse 10×12 resolution and with 4th order Legendre basis functions. The resulting x- and y-displacements are shown in Fig. 4.10. This perforated plate scenario is a common benchmark to test and verify elastic solid body solvers.

[10] similar to XFEM approach

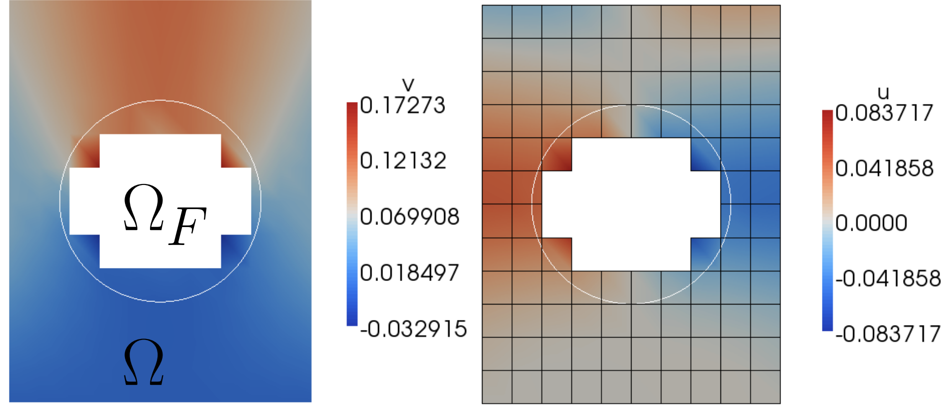

Figure 4.10.: One perforated plate benchmark with a 12×10 resolution. It shows (left) the vertical displacements v and (right) the horizontal displacements u. These computations were made with the Sundance toolbox [61].

4.2.4. Lagrange Multiplier Method

The Lagrange Multiplier Method (LMM) is a general approach for optimization under constraints and its formulation can be found in several textbooks (e.g., [21]). For the sake of completeness, we restate this formulation, where the main objective is to minimize a functional

$$J(u) = \frac{1}{2} a(u,u)_\Omega - \langle f, u \rangle_\Omega, \tag{4.10}$$

where $a(\cdot,\cdot)_\Omega$ is a bilinear form and $\langle f, \cdot \rangle_\Omega$ is a linear functional, both defined previously in Chapter 2. In this case, the bilinear operator maps $M \times M$ to $\mathbb{R}$, with M as the associated Hilbert space. In addition to Equation (4.10), we consider a constraint of the minimization problem (4.10) in the form of a bounded bilinear form $b : M \times N \to \mathbb{R}$.

$$b(u,\mu)_\Omega - \langle g, \mu \rangle_\Omega = 0 \ \forall \mu \in N. \tag{4.11}$$

$\langle g, \mu \rangle_\Omega$ and $\langle f, u \rangle_\Omega$ are the dual operators, which are linear operators on the Hilbert spaces M and N (defined in Chapter 2).

Condensing the constraint (4.11) and the objective function (4.10) into one function results in the Lagrange function

$$\mathcal{L}(u,\lambda) = J(u) + \left(b(u,\lambda)_\Omega - \langle g, \lambda \rangle_\Omega\right). \tag{4.12}$$

The minimization of the functional $\mathcal{L}(u,\lambda)$ results in an optimal $u \in M$ and $\lambda \in N$, where in the literature λ is called Lagrange multiplier. Similar to Theorem. 2.1.1 (Lax-Milgram), the minimization problem of (4.12) is equivalent with the following variational problem, where one needs to find $(u,\lambda) \in M \times N$ such that

$$\begin{aligned} a(u,v)_\Omega + b(v,\lambda)_\Omega &= \langle f, v \rangle_\Omega, \forall v \in M \\ b(u,\mu)_\Omega &= \langle g, \mu \rangle_\Omega, \forall \mu \in N. \end{aligned} \tag{4.13}$$

After discretization, the variational problem of equations (4.13) results in a linear system of equations. This linear system of equations is solvable, only if the so-called *inf-sup* condition is satisfied, which usually implies that M and N need to be discretized differently in space (e.g., different order basis or different types of basis functions). The *inf-sup* condition can be circumvented by adding a stabilization term (e.g., $\int_\Omega \alpha^2 (\lambda\, \mu)\, dx$) to the second equation of (4.13). For more theoretical insights we refer to [21, 50].

The LMM formulation is general and widely used to impose boundary conditions weakly, with these boundary conditions contained in the constraint operator $b(\cdot,\cdot) : M \times N \to \mathbb{R}$. Due to the generality of the LMM for IB conditions, this has been a research topic in numerous publications, and this method was already proposed in the early 70s [3] and is still used nowadays [35, 95]. For imposing a Dirichlet BC, a *classical* LMM has a more concrete form, where only the bilinear operator a remains general. In the following, the Dirichlet boundary condition on $\partial\Omega$ is $u|_{\partial\Omega} = g$ and the resulting system is

$$\begin{aligned} a(u,v)_\Omega + (v,\lambda)_{\partial\Omega} &= \langle f, v\rangle_\Omega \,, \forall v \in M \\ (u-g,\mu)_{\partial\Omega} &= 0 \;\,, \forall \mu \in N. \end{aligned} \tag{4.14}$$

In (4.14), the bilinear form b has been replaced by the boundary integral $b(u,\mu)_{\partial\Omega} = \oint_{\partial\Omega} u\mu\, dc$ and $\langle g,\mu\rangle_{\partial\Omega} = \oint_{\partial\Omega} g\mu\, dc$. It is important to note, that λ in (4.14) is only needed to be defined on $\partial\Omega$. However, if λ is defined on whole Ω, this approach is called *distributed* Lagrange multipliers [42, 37]. An approach similar to the distributed LMM is presented in [95]. The obvious disadvantage of the distributed LMM is that it induces additional unknowns λ, which are usually discretized on the same mesh as the unknowns u. This results in a significantly larger system than the one with the original variational form. On the other hand, in the case of moving boundaries, this approach has some practical implementational advantages compared to the approach with only local λ on $\partial\Omega$.

Defining λ only on $\partial\Omega$ certainly has the advantage that the resulting discrete system is not significantly larger then the one resulting directly from the variational problem.[11] For this reason, this approach is more wide-spread than the distributed LMM. In order to apply directly Equation (4.14) with λ defined only at the boundary, one needs an $d-1$ dimensional interface mesh that represents $\partial\Omega$. The creation of such a mesh is quite costly, especially in 3D. Therefore, it is desirable to use the same mesh, which the unknown u is discretized on by defining λ on intersected cells only.

The LMM method is often combined with the XFEM that employs special enriched functions to capture the discontinuity at the boundary. Enriched functions have usually a jump to the zero value at the boundary, so they do not have support in the fictitious domain Ω_F. Due to these jumps, it is important that the solution function u is forced to the Dirichlet values g on $\partial\Omega$. For FSI problems, [64] and [35] present several XFEM and LMM approaches, where λ is defined only on the boundary. The mortar FEM with Lagrange multipliers is introduced by [10] that is also used in an FSI context.

[11] without LMM

4.2.5. Nitsche's Method

The application of Nitsche's method for various IB applications is the main contribution of this thesis. This method was introduced in [69] for the elliptic Poisson equation as a consistent method that does not require additional unknowns or a special interface mesh. However, it requires the value of a penalty coefficient, for which the optimal value is still topic of research.

The idea is to define and minimize a functional J, the so-called energy functional that penalizes both deviations from the solution of the PDE inside the domain and from the boundary conditions on the boundary. The main step is the determination of J such that the resulting analytical minimization formula becomes not just consistent but also as simple as possible.[12] The resulting Nitsche's formulation for a given PDE is a weak formulation, where in addition to the existing integral terms new boundary integrals appear. This methodology results in a consistent method to impose a boundary condition in a general way and without additional degrees of freedom.

Due to its generality, Nitsche's method is employed in mesh-free contexts [4, 47], and in mesh-based discretizations for interface [40, 39] and IB problems [28, 49]. However, we employ this method in this thesis only for IB problems. In the following, we introduce first Nitsche's method for the Poisson equation. Afterwards, we formulate Nitsche's method for the Stokes and Navier-Stokes equations.

Nitsche's Method for the Poisson Equation

Nitsche's method was introduced for the Poisson equation [69], and here, using this rather simple PDE, we present the main idea of the method. A more detailed mathematical derivation can be found in Appendix A.2.

We restate the Poisson equation in the strong form on a continuous domain Ω:

$$\begin{aligned} -\Delta u &= f \;\; in\, \Omega, \\ u &= g \;\; on\, \partial\Omega. \end{aligned} \tag{4.15}$$

Correspondingly, we have the weak form of the equation, with the non-vanishing boundary terms and with the solution function $u \in V$ in the Hilbert space V

$$-\int_{\Omega} \Delta u\, v dx = \int_{\Omega} \nabla u \nabla v dx - \oint_{\partial\Omega} \left(\nabla\, u \cdot \mathbf{n}\right)\, v dc = \int_{\Omega} f\, v dx, \;\; \forall v \in V.$$

The next and crucial step is to define the energy functional $J\,(u)$ as it is stated in [69]

$$J\,(u) \;\; = \;\; \int_{\Omega} u_x^2 + u_y^2 dx - 2 \oint_{\partial\Omega} u\, u_n - \psi \oint_{\partial\Omega} u^2, \tag{4.16}$$

[12] With a different J, the resulting formula could be several lines long and could pose a significant implementational overhead.

with a shorter notation of the derivatives u_x, u_y, and u_n. The last term in (4.16) is a stabilization term with ψ as a penalty coefficient. For this coefficient, it is required that $\psi \to \infty$ with the mesh width $h \to 0$ becoming finer. $J(x)$ is used as a measure for the difference between the exact solution u_A and our approximate solution u. The objective is to minimize this norm, which gives rise to the minimization problem

$$J(u_A - u) = \min_{w \in V_h} J(u_A - w). \tag{4.17}$$

We transform $J(u_A - u)$ to the following form (see Appendix A.2)

$$J(u_A - u) = J(u_A) + J(u) - 2\int_\Omega f u dx + 2\oint_{\partial\Omega} (g\, u_n - \psi u) dc.$$

Next, we derive this equation with respect to u and set it to zero to achieve the equation form. Using the notation for the discrete basis functions $u = \sum_E y_E \varphi_E,\ y_E \in \mathbb{R}$, where the unknowns are in the vector $\mathbf{y}$. To find the correct discrete solution $\mathbf{y}$, we have to set up the derivative with respect to this vector

$$\begin{aligned} \frac{1}{2}\frac{\partial J(u_A - u)}{\partial y_E} &= \int_\Omega u_x \varphi_{E,x} + u_y \varphi_{E,y} dx - \oint_{\partial\Omega} u_n \varphi_E dc - \oint_{\partial\Omega} u \varphi_{E,n} dc \\ +\psi \oint_{\partial\Omega} u \varphi_E dc &- \int_\Omega f \varphi_E dx + \oint_{\partial\Omega} g(\varphi_{E,n} - \psi \varphi_E) dc = 0. \end{aligned} \tag{4.18}$$

Equation (4.18) is the resulting Nitsche's formulation presented in [69, 49]. The first two terms and the fourth term in (4.18) represent the original weak form of the Poisson problem. In addition, Nitsche's formulation contains two the penalty and two boundary integral terms. For a detailed derivation of Nitsche's method, we refer to Appendix A.2.

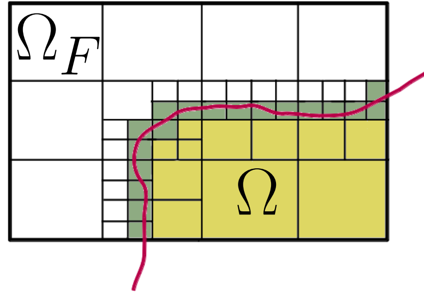

Figure 4.11.: Domains Ω and Ω_F on a Cartesian mesh. The cells intersected by the boundary (gray-green) need to be treated specially.

Nitsche's method for the Poisson equation contains two types of terms, volume integrals and boundary integrals. The implementation of this method only requires the computation of these two types of terms. The penalty factor is set as $\psi = \frac{C}{h}$, where $C \in \mathbb{R}$ and h is the mesh width at the boundary. This assures, that the required condition for consistency holds as $h \to 0$ so $\psi \to \infty$.

In the context of Cartesian meshes as illustrated in Fig. 4.11, evaluating these volume and boundary integrals implies a special numerical approach. The cells that are intersected by the boundary need to be treated separately for the volume integrals, such that only the part in Ω will be considered. The boundary integrals on $\partial\Omega$ with an underlying Cartesian mesh also pose a similar challenge. Since the boundary geometries can have an arbitrary shape, the first step is to use a given discretization of the boundary within an intersected cell. For consistency reasons, it is important to use the same boundary discretization for both boundary and volume integration. All the methods, corresponding implementations, and the boundary discretization used for this thesis are described in Chapter 7.

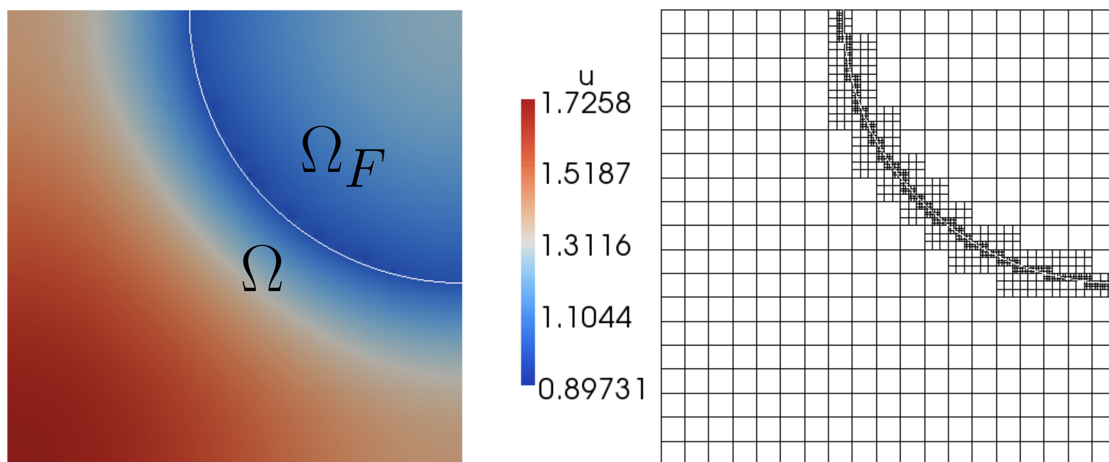

Figure 4.12.: 2D example of the Poisson equation with the IB conditions imposed by Nitsche's method (4.18). The boundary is represented by a circle arc, colored with white (left), Ω_O is the unit square. By refining the mesh at the boundary (right) up to level two, we achieve higher accuracy. The computations were made with the Sundance toolbox [61].

We illustrate the generality and the strength of Nitsche's method by considering two examples of the Poisson equation. The first example is a 2D scenario with a constant Dirichlet boundary condition $g = 1$ on the IB. In this concrete case, the *immersed domain* Ω_O is the unit square and the IB is represented by a circle as illustrated in Fig. 4.12. On Ω, we solve the Poisson problem $-\Delta u = 3$ with the mentioned BC on the circle. The BC is imposed with Nitsche's method (4.18). Along the boundary, the Cartesian mesh is refined up to level two, while keeping the 1-irregularity condition.[13] The penalty coefficient is set to $C = 6.0$. Fig. 4.12 shows the resulting solution u and the underlying adaptive Cartesian mesh.

Previously, we defined the immersed domain as $\Omega_O = \Omega \cup \Omega_F$, where $\Omega \cap \Omega_F = 0$. In this

[13]See Chapter 6

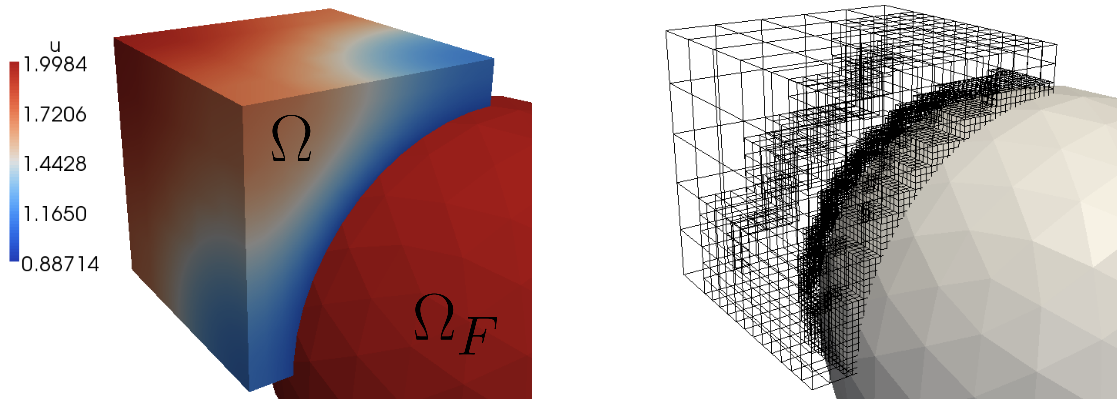

Figure 4.13.: 3D example of the Poisson equation with the IB conditions imposed by Nitsche's method (4.18). The boundary is represented by a sphere, colored with red (left), Ω_O is the unit cube. Similar to 2D, we refine the mesh at the boundary (right) up to level two. The computations were made with the Sundance toolbox [61].

example we have cells that are completely contained in Ω_F. These cells can be either ignored, or we can solve the same Poisson equation there weighted by a small factor, e.g., 10^{-8}. The first option certainly makes more sense for a stationary geometry. On the other hand, for moving boundaries, the second option, from the implementational perspective, has numerous advantages since allocation and deallocation of DoFs on these cells is avoided, while the boundary is moving. These aspects of Nitsche's method will also be discussed in Chapter 8.

The next example is set up in 3D, but the equation is left unchanged $-\Delta u = 3$. In this case, the sphere represents the two-dimensional boundary $\partial\Omega$ that intersects the cells in 3D. The solution function u with the underlying mesh is shown in Fig. 4.13. The penalty coefficient is set to $C = 50$.

The examples visualized in Fig. 4.12 and Fig. 4.13 demonstrate the generality and strength of Nitsche's method. In contrast to the Lagrange multiplier method, this method does not require additional DoFs, and geometry based enrichment of the basis functions is also not necessary. Since the boundary cells have the same basis functions as the rest of the cells in Ω, Nitsche's method enables us to use the same discretization on the whole mesh. As most IB methods, Nitsche's method requires a geometry representation that is used to compute the necessary volume and boundary integrals. The only parameter that needs to be set is the penalty coefficient C, which value can be determined experimentally. These key properties allow for the usage of Nitsche's method in a toolbox environment, which is one of the main tasks of this thesis.

At last, we show a simple convergence analysis of the Nitsche Method for a simple

scenario in order to prove that even with Cartesian meshes one can achieve second order accuracy[14] on the boundary. Chapter 7 contains the algorithms and methods that allow for the usage of the Nitsche Method within Sundance. For the convergence analysis, we considered the simple scenario on Fig. 4.14, where on the unit square a circle with a radius of 0.23 is placed at the position $(0.51, 0.491)$.[15] This Dirichlet boundary consists of a polygon with considerable higher resolution than the mesh. We measure the L_2 error on this polygon with respect to the homogeneous Dirichlet boundary condition. The measured L_2 errors are shown in Tab. 4.1, which are measured for different mesh resolutions, for Q_1, and Q_2 basis functions.

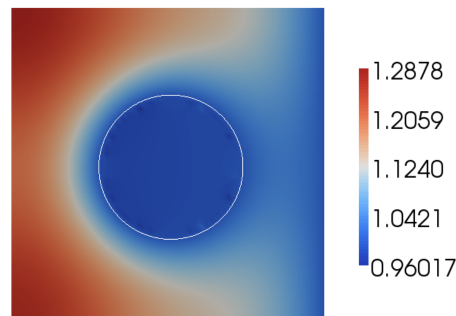

Figure 4.14.: Scenario of the convergence analysis. We impose with Eq. (4.18) and $\alpha = 50$ a homogeneous Dirichlet BC on the circle and we measure the L_2 error on Γ between this homogeneous value and the resulted solution. Chapter 7 describes the concrete implementation of Eq. (4.18).

regular Cartesian Mesh	L_2 error with Q_1	L_2 error with Q_2
7×7	2.683e-2	2.543e-3
14×14	1.083e-2	6.941e-4
28×28	8.207e-4	4.175e-4
56×56	4.257e-4	1.650e-5
112×112	1.258e-4	6.466e-6
224×224	1.232e-5	4.023e-6
Measured order	2.22	1.86

Table 4.1.: The L_2 error measured on Γ (the circle on Fig. 4.14) with increasing regular mesh resolution. In the last line, we represent the measured average order of the Nitsche Method. The measured order of the error in both cases is around two.

The measured error reduction orders in Tab. 4.1 show that using a regular Cartesian mesh and the Nitsche Method one can achieve second order accuracy at the boundary, which

[14] as with unstructured meshes
[15] In order to avoid symmetrical error cancellations.

is our main argument for this IB method. In the classical way, only an unstructured mesh has such order[16] on the boundary.

Nitsche's Method for the Stokes and Navier-Stokes Equations

In this subsection, we introduce Nitsche's method for the Stokes and Navier-Stokes equations. The idea for the derivation was shown already for the Poisson equation, therefore, we just state the method here. A detailed derivation of Nitsche's formula for the Stokes equations is described in Appendix A.3.

Next, we restate the stationary Navier-Stokes equations in the form that is used previously in [19] and in Eq. (3.5) and (3.6) of Section 3.1.

We consider the computational domain $\Omega \subset \mathbb{R}^d$ with $d = 2, 3$ and the unknown functions $u = (v, p) \in H^1(\Omega)^2 \times L_0^2(\Omega)$ with the velocities v and the pressure p. Further, we denote the boundary of Ω as $\Gamma = \partial\Omega$ and $g \in H^{1/2}(\Gamma)$ is the Dirichlet boundary condition. $f \in L^2(\Omega)^2$ represents external forces acting on the fluid. In the following, for the sake of simplicity, we consider $\rho^f = 1$. ν represents the kinematic viscosity of the fluid. With these definitions, the Navier-Stokes equations read

$$
\begin{aligned}
-\nu\Delta v + (v \cdot \nabla)v + \nabla p &= f \;\; in\, \Omega, && (4.19)\\
\nabla \cdot v &= 0 \;\; in\, \Omega, && (4.20)\\
v &= g \;\; on \;\; \Gamma. && (4.21)
\end{aligned}
$$

The transformation to the weak form of equations (4.19)-(4.21) has already been presented in Section 3.1 (see Equation (3.7)). We use the same definition for the test functions except they do not have compact support on Γ. In contrast to Section 3.1, the resulting boundary integrals do not vanish:

$$
\begin{aligned}
(-\nu\Delta v, \psi)_\Omega &= \nu \int_\Omega \nabla v : \nabla \psi \, dx - \nu \int_\Gamma \partial_n v \psi \, dS(x)\\
&= \nu \left(\nabla v, \nabla \psi\right)_\Omega - \nu \langle \partial_n v, \psi \rangle_\Gamma,\\
(\nabla p, \psi)_\Omega &= -\int_\Omega p \left(\nabla \cdot \psi\right) dx + \int_\Gamma p\, n \cdot \psi \, dS(x)\\
&= -\left(p, \nabla \cdot \psi\right)_\Omega + \langle p\, n, \psi \rangle_\Gamma.
\end{aligned}
$$

Summing up all volume integrals yields the functional a

$$
a(u, \phi) := \nu \left(\nabla v, \nabla \psi\right)_\Omega + \left((v \cdot \nabla)v, \psi\right)_\Omega - \left(p, \nabla \cdot \psi\right)_\Omega + \left(\nabla \cdot v, \xi\right)_\Omega ,
$$

whereas the boundary integrals are denoted by c

$$
c(u, \psi) := -\nu \langle \partial_n v, \psi \rangle_\Gamma + \langle pn, \psi \rangle_\Gamma .
$$

[16]only with linear transformation

These terms give the weak formulation of the stationary Navier-Stokes equations

$$a(u, \phi) + c(u, \psi) = (f, \psi)_\Omega \ \ \forall \ \phi, \tag{4.22}$$

where the Dirichlet boundary condition is not enforced yet.

In the following, we present Nitsche's method incrementally in order to enforce the Dirichlet conditions weakly. Using Nitsche's method implies adding penalty-like terms and terms that maintain the skew-symmetry of the Stokes operator [15, 41]. The skew symmetric counter term $\hat{c}$ of c is

$$\hat{c}(v, \phi) := -\nu \langle \partial_n \psi, v \rangle_\Gamma - \langle \xi n, v \rangle_\Gamma.$$

In the next step, we consider the discretized velocity and pressure space. To fulfill the *inf-sup* criterion we use different discretization spaces for velocity and pressure. With this specific choice, no stabilization terms are required, which were presented in Section 3.1. In the case of *inf-sup* unstable elements, the stabilization adds only the stabilization-terms to Nitsche's method as shown in [15, 41]. Further, the penalty terms $\nu \frac{\gamma_1}{h} \langle v, \psi \rangle_\Gamma + \frac{\gamma_2}{h} \langle v \cdot n, \psi \cdot n \rangle_\Gamma$ are also added to the weak form.

In [15], additional inflow stabilization terms are considered, namely $-\langle (v \cdot n)^- v, \psi \rangle_\Gamma$, where $(t)^- = \min\{t, 0\}$. Compensating this term, [15] adds $-\langle (g \cdot n)^- g, \psi \rangle_\Gamma$ on the right side. In our numerical examples (see Chapter 7), it turned out that this inflow stabilization is negligible. Thus, in the following, we will not consider this additional inflow stabilization.

Summing up all the listed terms gives rise to the Nitsche's method of the stationary Navier-Stokes equations (4.22), where we denote the discrete velocity space as V_h and the bilinear finite element discrete pressure space is Z_h:

$$\begin{aligned} &a(u_h, \phi_h) + c(u_h, \psi_h) + \hat{c}(v_h, \phi_h) + \nu \frac{\gamma_1}{h} \langle v_h, \psi_h \rangle_\Gamma + \frac{\gamma_2}{h} \langle v_h \cdot n, \psi_h \cdot n \rangle_\Gamma \\ &= (f, \psi_h)_\Omega + \hat{c}(g, \phi_h) + \nu \frac{\gamma_1}{h} \langle g, \psi_h \rangle_\Gamma + \frac{\gamma_2}{h} \langle g \cdot n, \psi_h \cdot n \rangle_\Gamma \ \ \forall \ \phi_h \in V_h \times Z_h. \end{aligned} \tag{4.23}$$

Here, h denotes the local mesh size on the boundary Γ. The formulation (4.23) is consistent in the sense that the solution of the Navier-Stokes equations satisfies the variational problem. Further, convergence in the case of the Stokes equations (without the nonlinear convective terms) is also assured [19].

Similar to Equation (4.23), [28] defines also a Nitsche's formula for the advection-diffusion and Navier-Stokes equations in the context of IBs that does not have all the terms presented in (4.23). Although formulation (4.23) of Nitsche's equation is the same as the formulations in [15, 41], our approach[17] still appears to be unique for the Navier-Stokes problem. In both previous works [15, 41], Nitsche's method is applied to boundary conforming meshes (in 2D on the edges of the cells), where the BC could be imposed

[17] published in [19]

in a classical way. We apply (4.23) to general non-conforming cases, where the cells' boundaries do not represent Γ. In this thesis, we also show that this formulation can be applied not just to IB scenarios, but can also be extended to transient scenarios with moving geometries. For such transient cases, the formulation (4.23) is transformed to a parabolic one by adding the time derivative of the velocity $\frac{\partial u}{\partial t}$. The proposed approach is validated in Chapter 7 and Chapter 8 with 2D and 3D benchmark calculations, where consistency on the boundary plays a special role in order to compute the correct forces acting on the obstacle.

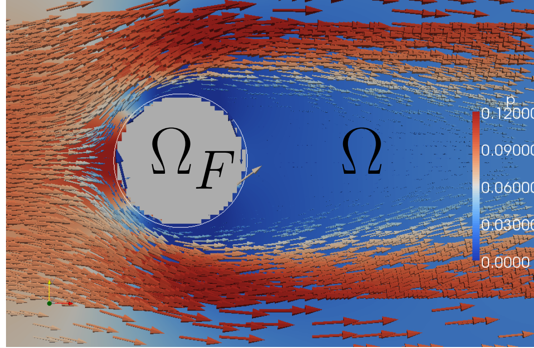

Figure 4.15.: Nitsche's method for the stationary Navier-Stokes equations applied to the 2D-1 scenario in [80]: The obstacle is represented by the white circle. The figure shows the pressure and the velocity fields. The underlying mesh is an adaptive Cartesian, that is refined on the boundary.

Analog to Nitsche's method for the Poisson equation, formulation (4.23) requires only the implementation of volume $(\cdot,\cdot)_\Omega$ and the corresponding boundary integrals $\langle \cdot{,}\cdot \rangle_\Gamma$. In contrast to LMM and XFEM, no interface mesh or additional DoFs are required on Γ. The only parameters that need to be determined are γ_1 and γ_2 and in our case will be chosen experimentally. In [15], these values were set to $\gamma_1 = 3,\ \gamma_2 = 0.1$, and [41] used slightly higher values $\gamma_1 = 10, \gamma_2 = 1$, whereas both imposed the Dirichlet BC on the edge of the cells in 2D. In our computations on IBs (see Chapter 7 and Chapter 8), we set these parameters with values between 10^2 and 10^4.

To highlight in advance the usability of Nitsche's method (4.23) for the Navier-Stokes equations, we consider a two-dimensional example in Fig. 4.15. The underlying mesh for the scenario is an adaptive Cartesian mesh. Fig. 4.15 also illustrates the boundary of Ω, which is a white circle. The boundary cells, which are cut by the circle are also shown in Fig. 4.15. In Chapter 7 and Chapter 8, we describe the methods in our toolbox, which are used in order to use Nitsche's method in a general and toolbox manner.

5. Sundance PDE Toolbox Introduction

The focus of Chapter 2 is the mathematical foundation of the FEM. From this general mathematical approach, results a general method to discretize and solve a given PDE. The introduced mathematical notions in that chapter are the key concepts to understand the software architecture of a FEM-based PDE toolbox. In this chapter, we mainly present the software architecture of the FEM-based PDE toolbox Sundance [61, 60, 62] that is also found in other similar PDE toolboxes, which we extend with various new capabilities in the next chapters in order to compute problems with IBs. We want to emphasize here that the described structure in this chapter does not include the developments that are the results of this thesis. Our goal, in this chapter, is to introduce the base line architecture that we had to our disposal. In the following chapters, we introduce our developments incrementally that we added modularly to the Sundance PDE toolbox. In Chapter 2, we outlined the spatial discretization, linear system assembly and the imposition of Dirichlet BCs that are modular parts of the simulation process and are present in most of the FEM-based PDE toolboxes. By presenting the simulation process in this section, we focus on the modular components and the interfaces between them. These interfaces we also described in [18]. The underlying software engineering aspects and solutions are not discussed here, but we demonstrate the capabilities and the high-level descriptive language of Sundance. In the last section, we present an overview of other existing open-source PDE toolboxes, which are currently available. We compare the actual capabilities[1] of Sundance to other similar toolboxes.

5.1. Structure of the Sundance PDE Toolbox

The last section of Chapter 2 introduces the necessary mathematical concepts for a general FEM approach to setup a linear system of equations based on the element basis function, mesh discretization, and the weak formulation of the PDE. There are several steps between the continuous weak problem formulation and the final discrete solution. The collection of these numerical methods we call as the **simulation pipeline**. In the following, we present the simulation pipeline in the Sundance toolbox that is composed

[1] Not including the contribution of this thesis

of several stages. This pipeline starts with the PDE problem formulation that is the essential starting point for the pipeline as illustrated in Fig. 5.1.

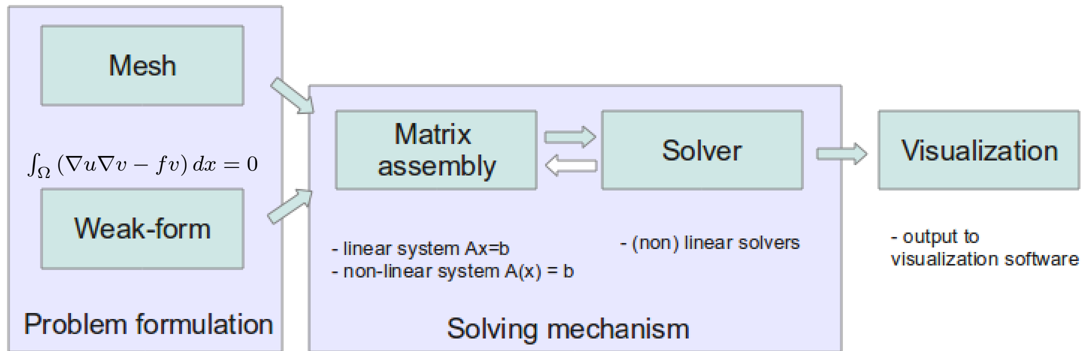

Figure 5.1.: Illustration of the Sundance PDE toolbox's simulation pipeline, representing the sequence of methods that result in the solution and visualization of the PDE problem. The interaction with the user takes place in the first stage that is called 'Problem formulation'. 'Solving mechanism' is more decoupled from user interaction and forms the major part of the computational load. At the final stage of the pipeline, visualization and evaluation of the numerical results takes place.

At the problem formulation stage, the user specifies the problem in the given frame of the PDE toolbox. In this phase, it is crucial that Sundance's formulation language offers an efficient way to describe a given PDE problem and also specifies the problem uniquely. The following two stages we name as the solving mechanism that uses the FEM discretization to set up the matrix and an external solver to solve the system. The final stage of the pipeline represents the visualization and the evaluation of the numerical results. These stages are present not just in Sundance but in most commercial and non-commercial FEM-based PDE toolboxes. Sundance offers a high-level descriptive language to define a PDE problem in weak form and this was the main reason why this toolbox was chosen as the software basis for our developments. The following sections describe the stages of the simulation pipeline that mathematically were already introduced in Chapter 2. Therefore, we do not insist here on the theoretical aspects of each step, but on the modularity of the simulation pipeline.

5.1.1. Problem Formulation

The first step in the simulation of a given PDE with a FEM-based toolbox is to input the problem specifications into the software (see Fig. 5.1). In the case of mesh based FEM, this implies several information. First is to define $\Omega \subset \mathbb{R}^d$ that represents the computational domain. The partitioning of Ω in discrete cells is done by the mesh generation

that results in the computational mesh of the PDE problem. This process represents a crucial part of the solving process. Especially for computations on distributed memory systems, the partitioning of the mesh is important, as we discussed in Chapter 4. Prior to our developments, Sundance had an interface only to parallel simplex unstructured meshes in 2D and 3D and the interface to this type of mesh is described next.

Mesh Interface

The mesh implicitly defines not just the computational domain Ω, but also the spatial resolution. In the literature, the mesh resolution is also called *h-refinement*. In Chapter 4, we mentioned that unstructured meshes represent the most general structure that a computational mesh can have. Therefore, they allow for the arbitrary positions of nodes, such that the cells can have any affine shape[2] and are able to represent complex boundaries as well. For this reason, the mesh interface of Sundance is a general one and fits the structured mesh case as well.

Some of the aspects of the structured and unstructured meshes were already discussed in Chapter 4. Here, however, we define the mesh components and their mathematical properties in order to have a well defined interface. Since Sundance is using the mesh only for geometrical position and connectivity information, we define first the geometrical entities that form an unstructured mesh. These definitions are aligned with the interface notation that is used within Sundance. The term *mesh entity*, denoted with E, represents all geometrical entities forming a mesh. Each mesh entity has an associated dimension: a node (or point) by definition has dimension zero, a line dimension one, a triangle or rectangle dimension two and a tetrahedron or a brick dimension three. We further denote $dim\,(E)$ as the function that returns the dimension of the entity E. The dimensionality of the problem is given by $N = \max_E\,(dim(E))$, where $N \in \{2, 3\}$ for our problems. I_d is the set of mesh entities with dimension d. Entities with the maximal dimension N are called, analog to Chapter 4, as *cells*.

Definition 5.1.1 *A **facet** F is a d-dimensional mesh entity that is part of a D-dimensional entity E with $d \geq 0$, $d < D \leq N$. We denote the facet F of mesh entity E as $F \subset E$.*

The intersection of two mesh entities E_i and E_j might contain facets with different dimensions. In such cases, we consider all the possible common facets as the intersection. We write the intersection operation as $F = E_i \cap E_j$, where $i \neq j$, $E_i, E_j \in I_d$. This inclusion of one low-dimensional mesh entity into high-dimensional mesh entities can be also defined in the opposite direction in the next definition.

Definition 5.1.2 *A **co-facet** F is a D-dimensional entity that contains the d-dimensional facet E with $D \geq 0$, $d < D \leq N$. The entity F is a co-facet of E, so we write $E \subset F$.*

[2] Isoparametric case is not considered here.

The number of co-facets of the $(N-1)$-dimensional facets is an important information to determine boundary facets, which are needed to impose the boundary conditions. Each $N-1$ dimensional facet, which has only one co-facet, is a boundary facet. We name the co-facets of highest dimension N as *maximal co-facets*.

With the definitions of facets and the intersections of cells we can establish the definition of hanging facets that was already mentioned in the context of Cartesian meshes.

Definition 5.1.3 *A given facet $F = E_i \cap E_j, E_i, E_j \in I_d$, $0 \leq d < N$ is hanging if and only if $F \not\subset E_i$ or $F \not\subset E_j$. Then, we call this facet as **hanging facet**.*

One well-known example for hanging facet is the hanging node, which is a zero dimensional facet. Besides the hanging nodes, in three-dimensional adaptive Cartesian meshes one can face hanging lines and hanging quads. Obviously, the intersection of two neighboring cells with different refinement levels in a tree-structured mesh (see in Chapter 4 Fig. 4.2 and Fig. 4.2) is a facet only of the cell with the higher refinement level and this facet is a hanging facet.

Definition 5.1.4 *A mesh is called **0-irregular (conforming)** if it does not contain hanging facets.*

Having a 0-irregular (conform) mesh simplifies the numerical approach to ensure C^0-continuity between the elements that is required in most problems. Therefore, the original mesh interface in Sundance does not threat the 1-irregular case. Using the defined components above, we define the 0-irregular unstructured mesh interface, which is composed of several functions. The distributed parallel aspects of the interface are discussed later in this section.

1.) *Mesh entity's identification number(ID)*: This function implements the most important requirement for a Sundance mesh that each E entity belonging to $E \in I_d$ must have its own unique positive ID. These numbers start from zero till the number of entities in I_d. From this follows, that a given facet or cell E is identified by its dimensionality d and by its ID.

2.) *Node's position*: The function provides two- or three-dimensional coordinates of a given node (defined by the node's ID). The location in $\mathbb{R}^d$ of a specified entity can be determined by the positions of its nodes. These node's coordinates are also used to create the cell's Jacobian matrix, a functionality that is needed in the system matrix assembly process as shown in Chapter 2.

3.) *Mesh entity's facets*: This function provides all the facets of a specified dimension for a given entity (specified by the entity ID and the dimension d). The number of facets for a given dimension d_E of co-facet and facet dimension d_F is constant. The functionality returns the facets, specified by their IDs.

4.) *Mesh entity's maximal co-facet and the facet's index inside the co-facet*: Returns the number of maximal co-facets for a d-dimensional facet ($d < N$) and the index of the facet in the list of facets of these maximal co-facets. The facet's index within a maximal co-facet is crucial information to determine the DoFs within a cell that are located on the facet.

The functionalities above represent the serial unstructured mesh interface that Sundance uses. These four functions provide enough information in the serial case to setup the problem's system matrix. The mesh is only used for geometry and connectivity information of the different dimensional mesh entities.

In the following, we focus on the parallel aspects of the mesh interface. Sundance also facilitates simulation on distributed memory systems. Efficient parallelism on such machines prohibits the central storage of the mesh, since the individual processes communicate with each other by sending direct messages instead of sharing a common memory. In Chapter 4, we discussed the necessity of domain decomposition in order to distribute the computational load among processors. In most cases, the computational load is direct proportional to the number of cells.

Being at the first stage of the simulation pipeline in Fig. 5.1, and similar to other toolbox approaches (e.g., [6]), the mesh is the component that drives this parallelism. Thus, it is a crucial factor for the parallel efficiency. Since no central data storage is allowed, Sundance relies solely on the mesh to distribute the cells among the processes in a load-balanced manner.

In such parallel cases, the system matrix is generated distributed on the system, where each process only has the assigned lines of the matrix. In order to assembly this matrix without additional communication, Sundance requires the mesh to have a *ghost cell* layer at the inter-process boundary, such that the assigned system matrix lines can be computed independently. Ghost cells are cells at the process boundary, which are parts of the locally stored mesh in a process, but they belong logically to another process. For 0-irregular (conform) meshes, the ghost cells are direct neighbors of cells that belong to the local processor, i.e., a ghost cell always has a common facet with a local cell as illustrated in Fig. 5.2. Regarding the ownership of the mesh entities, it is important to mention, that the policy is, that the entities located on the process boundary belong to the process with the smaller rank.

In order to incorporate this parallel functionality into the mesh interface, first, there is a need to identify each mesh entity uniquely both in a global sense regarding the complete mesh and in a local sense regarding the mesh partition of each process. For this reason, all facets and cells have both its own IDs in the global mesh and a local index in the local mesh structure. In Fig. 5.2, the global and local meshes are also illustrated, where the global mesh is not stored centrally. In the following, we denote the ID of an entity in the local mesh as local identification number (LID) and the ID of an entity in the global mesh as global identification number (GID). The GIDs and the LIDs are positive

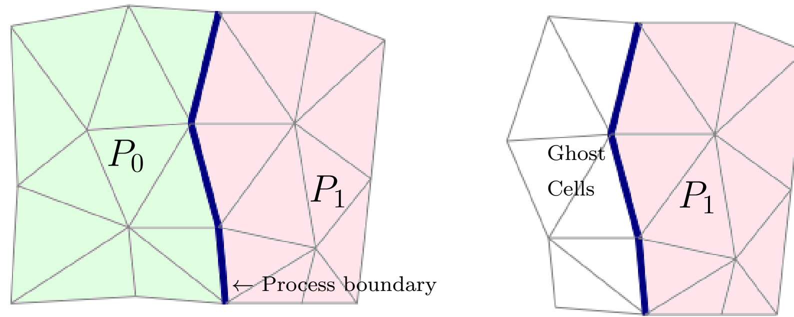

Figure 5.2.: Illustration of the ghost cells at the inter-process boundary. The thick blue line represents the boundary between the two processes P_0 and P_1 (left). It illustrates (right) the necessary ghost cell layer needed by process P_1 and also the local mesh of process P_1. The entities located on the process boundary belong to the process with the smaller rank.

integer numbers, and in serial computations it holds for all entities that LID=GID. For these reasons, the mesh interface contains additional functions for the parallel case.

1.) *mesh entity owner*: Function that returns the processor number which the specified entity (logically) belongs to. It is important to note that inside a cell not all the facets need to have the same owner.

2.) *local ID map to global ID*: Function that maps the LID of an entity to the GID, where the following relation holds $GID = OFF(p, d) + LID$.

3.) *global ID map to local ID*: Function that maps the GID of an entity to the LID, where the same relation holds $GID = OFF(p, d) + LID$.

In the relations above $OFF(p, d)$ represents the offset that depends only on the dimension d of the mesh entity and the local number (rank) p of the processor. These three interface functions assume that other components of Sundance are calling these functions with a GID that exists in the local mesh in parallel case.

The functions defined for the sequential case use the LID. The GIDs are used only in the last two functions. The seven functions defined in this section form the mesh interface within the Sundance toolbox and this interface is used by all the other components to interact with the mesh.

In the following, we highlight the problem descriptive language that creates the mesh. Sundance's descriptive language is C++ and the toolbox is implemented also in the same

language. The user specifies the mesh by defining a C++ object as shown in Code 1. Unstructured meshes are usually created by external mesher softwares and are saved into

Code 1 Creation of two meshes, one from external file and the second internally on the unit square.

```
MeshType meshType = new BasicSimplicialMeshType();
MeshSource meshReader =
 new TriangleMeshReader("meshInputFile.1", meshType);
MeshSource mesher =
 new PartitionedRectangleMesher(0,1.0,20,1,0,1.0,30,1,meshType);
Mesh meshExternal = meshReader.getMesh();
Mesh meshInternal = mesher.getMesh();
```

files. In the parallel case, the mesher also partitions the mesh into a specified number of processes. Sundance can also access such parallel meshes, which were generated externally and saved partitioned into a standard format. The example of Code 1 shows how a simplex mesh can be created inside Sundance, by using the high-level C++ objects. In the second line, an unstructured simplex mesh ($BasicSimplicialMesh$) is created from an external file. In the following line, a similar mesh is created but in a structured way.[3] In serial and parallel cases, the Sundance code is the same and parallelism does not require any additional interaction from the user that shows the true high-end toolbox potentials of Sundance. An unstructured mesh is usually created by an external mesher tool (e.g., CUBIT [65], ShowMe [83]) and then stored in various serial and parallel file format (e.g., exodusII [32], NetCDF [94]).

Weak Form and Problem Definition

The next part of the problem definition is the formulation of the weak form for a given PDE. First component of these weak forms is the domain, where a specific weak form is defined on. As an example, we consider the weak form of

$$\int_{\Omega} (\nabla u_h \nabla v_h - f v_h)\, dx = 0,\ \forall v_h \in V_{T,h} \tag{5.1}$$

where $V_{U,h}, V_{T,h} \subset H^1(\Omega)$, $u_h \in V_{U,h}$, and $f \in \mathbb{R}$ being a constant. The domain $\Omega \subset \mathbb{R}^d$ is defined by the cells of the mesh. Therefore, the first step is to define the collection of cells (or facets), where the integral is defined on. This first component, the collection of mesh entities, is called **cell-filter**. Technically, this implies that all mesh entities for a given dimension are selected in the first instance, then these entities are passed through a filter and only the entities that fulfill the filter condition are finally selected for the

[3] In this case, the code creates a 20×30 mesh on the unit square.

iteration. We denote the set of these selected cells for our example as I_Ω. Consequently the integral is given by a sum of integrals over cells or facets $E_i \in I_\Omega$,

$$\sum_{E_i \in I_\Omega} \int_{E_i} (\nabla u_h \nabla v_h - f v_h)\, dx = 0. \tag{5.2}$$

The condition for the filtering may vary from case to case. In the classical case, Ω is the whole computational domain and is covered by the cells of the created mesh. But for the lower-dimensional entities only those facets are selected, which a specified BC is imposed on. Code 2 shows the declaration of different cell-filters that are later used in the weak form declarations.

Code 2 Declaration of two cell filters. The cell filter *Omega* includes all the cells, whereas *Gamma* includes only the boundary facets that satisfy the condition specified in *GammaTest* class.

```
CellFilter Omega = new MaximalCellFilter();
CellFilter Boundary = new BoundaryCellFilter();
CellFilter Gamma = Boundary.subset(new GammaTest());
```

The next step is to specify the discrete test $V_{T,h}$ and unknown $V_{U,h}$ spaces. In Chapter 2, we introduced the Lagrange basis functions, which we mainly used in our applications. In the same chapter, the Ritz-Galerkin approach is also introduced, which implies $V_{T,h} = V_{U,h}$. For our concrete example (5.1), we use first order Lagrangian basis functions that are defined in Code 3. The mesh specifies the h-resolution of our discrete space, which combined with the chosen basis function form the discrete spaces $V_{T,h}$ and $V_{U,h}$.

In Equation (5.1), we also find the partial spatial derivatives of the test and unknown functions. This implies the declaration of spatial derivation operators that, during the weak form declaration, applied to the basis functions result in the spatial derivatives of the respective function. These type of *spatial derivative* operators are delivered by the $Derivative(i)$ class, where the index i specifies the dimension index in space. The basis functions inside Sundance are defined as polynomials, but their spatial derivatives are computed on the fly. This is enabled by an automated differentiation (AD) method implemented in Sundance.

The component of the weak form that is not visible in the mathematical form is the quadrature method. Even though the quadrature is needed for each cell integration only when the coefficients in the front of the integrals are space-variant.[4] For constant coefficients, the terms need to be integrated only on the reference cell (see Chapter 2) and for these cases, the required order can be computed dynamically. In most applications, the basis functions have the form of a polynomial that can be integrated up to numerical precision with a quadrature of order p, while the basis function has the same order. On the other side for non-constant coefficients, the specified quadrature method is used.

[4]In our case it is just constant 1.0 and $f = 3.0$.

Code 3 Weak form declaration of Equation (5.1). The first four lines declare the unknown and the test basis functions, which is followed by the gradient operator definition. After declaring the quadrature method, the weak form is condensated in one line. In the final line, we declare the Dirichlet BC $\oint_{\Gamma} (u_h - 1.0)\, v_h dc$ in a weak form.

```
Expr unknBase = new Lagrange(1);
Expr testBase = new Lagrange(1);
Expr u = new UnknownFunction( unknBase , "u");
Expr v = new TestFunction( testBase , "v");
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
QuadratureFamily quad = new GaussianQuadrature(2);
Expr weakForm = Integral( Omega , (grad*u)*(grad*v) - f*v, quad );
Expr bc = EssentialBC( Gamma , v*(u-1.0), quad );
```

The concrete use of the enumerated objects is illustrated in Code 3, where the Sundance's C++ objects are used for the weak form in Equation (5.1). Code 3 starts with the declaration of basis functions, which are the same for both test and unknown spaces. This is followed by the setup of the gradient operator that is a list of spatial derivation operators. The second last line holds the actual weak form in C++. One can easily recognize the similarity between Equation (5.1) and the corresponding C++ line. This visible correspondence between the mathematical formulation and the C++ code clearly demonstrates the high-level problem description capabilities of Sundance.

The last missing item from the well posed problem for our example is the Dirichlet BC,

$$\int_{\Gamma} (u_h - 1.0)\, v_h \, dx = 0, \; \forall v_h \in V_{T,h}, \; on\, \Gamma \tag{5.3}$$

that is formulated in a weak form. The imposition of boundary conditions formulated in such a way, was already discussed in Chapter 2, and their formulation is similar to a general weak form. However, this weak formulation has to be treated in a special way (row replacement with the results, see Chapter 2), therefore, in Sundance it is defined with the *EssentialBC* class. The last line of Code 3 shows the declaration of the Dirichlet BC on the selected boundary segment Γ.

At this point, we initiated the mesh and declared the weak form of the problem including the Dirichlet BC. The next step is to collect all these definitions into one object that will contain the problem description. Since the presented problem (5.1) is a linear one, we use the *LinearProblem* class as shown in Code 4. The first line defines such a *LinearProblem* object, where the input parameters are various objects defined previously. In second line, we illustrate how the solving mechanism is activated, with the created *solver* object. The result of this call is an expression that represents the final

solution.[5]

Code 4 Linear problem definition and solving.

```
LinearProblem prob(mesh, weakForm , bc, v , u , vecType);
Expr up = prob.solve(solver);
```

Sundance also has the capability to declare and to deal with nonlinear PDEs in the same general manner. In contrast to the linear example (5.1), we consider a nonlinear PDE (stationary Burgers equation)

$$\int_{\Omega} \left(\nabla u_h \nabla v_h - \left(u_h \cdot \nabla\right) u_v v_h\right) dx = 0,\ u_h \in V_{U,h} \forall v_h \in V_{T,h}, \tag{5.4}$$

where the unknown and test functions are scalar fields in 1D and the *grad* operator in Sundance is also defined accordingly. Regardless of the nonlinear term in the equation, the weak form (5.4) is directly transformed into Sundance code, as shown in the first line of Code 5. The connection between the mathematical formulation and the Sundance

Code 5 Nonlinear problem definition and solving.

```
Expr weakForm=Integral(Omega ,(grad*u)*(grad*v)+(u*grad)*u*v,quad);
DiscreteSpace UnknownSpace(mesh, List(unknBase), vecType);
Expr u0 = new DiscreteFunction(UnknownSpace, 0.0, "up");
NonlinearProblem prob(mesh, weakForm , bc, v , u, u0, vecType );
StatusType status = prob.solve(solver);
```

code is clearly visible in this case as well. There are several differences compared to the linear case. One is the declaration of this nonlinear problem with the $NonlinearProblem$ class. The solving of nonlinear problems requires the input of an initial (guess) value. This value is created in the second and third lines of Code 5 with the $u0$ object, which represents a discrete function with a global value of zero. The *DiscreteSpace* class enables the declaration of a discrete space formed by a collection of arbitrary finite elements. Sundance enables the use of such abstract but general objects that help the user to define the problem in a general and efficient way. The implementation of time stepping methods also require the use of the *DiscreteSpace* and $DiscreteFunction$ classes in order to store the solution from the earlier time steps.

In the last line of Code 5, the solution is computed, with the already created nonlinear solver object $solver$. The computed solution is placed in the $u0$ object, which before solving contained the initial solution. Aspects regarding the nonlinear solving mechanism are discussed in the following section.

[5]In case of convergence.

5.1.2. Matrix Assembly

In the next stage of the simulation pipeline, according to Fig. 5.1 and to the general approach presented in Chapter 2, the system matrix is set up. Although this approach is general and creates a modular process, where the solvers are easily replaceable, but the matrix entries' computation and storage poses a memory overhead. A so-called *matrix-free* approach would eliminate this overhead, but on the other side would significantly restrict the solver's modularity. This approach is not considered in this thesis and we only focus on the modular elements of the assembly process.

As first, we only consider linear problems, where the solution is directly given by the system matrix and the corresponding right-hand side vector. The nonlinear case implies in addition the computation of the Jacobian matrix and several *linear* solver steps, which are discussed at the end of this subsection.

Creating the matrix in the literature is named as the *matrix assembly* process, since the system matrix is assembled out of the facets' and cells' matrices. In Chapter 2, this assembly process was highlighted by the equations (2.12) and (2.13), where one entry $A_{i,j}$ of a matrix is computed by the sum over the cells that contain the i-th and j-th DoFs. In order to highlight the practical aspects of this assembly process, we consider the example in Fig. 5.3.

The simplex 2D mesh that is considered in Fig. 5.3, consists of 7 triangles and using a bilinear unknown and test basis functions, the discretized problem of our PDE-example (5.1) results in $A\,\mathbf{x} = \mathbf{b}$, with A as a square 7×7 matrix and the corresponding right-hand side vector $\mathbf{b}$. This linear system of equations is assembled out of the 7 cell integrals in an incremental way. In Fig. 5.3, we illustrate how the result of the weak form integral on cell with ID 7 (Cell 7) contributes to the system matrix A and vector $\mathbf{b}$.

In the first stage, the local stiffness matrix of Cell 7 is computed. In our case, this does not involve the actual numerical quadrature on this cell. Since no space variant coefficients are present, one can compute only once the integral and then for each cell apply the necessary Jacobian matrix transformation[6], as shown by Equation (2.14) in Chapter 2. This simplification is used also within the assembly process of Sundance, where Sundance calls the BLAS routines not just for one cell but for a group of cells. Therefore, the assembly process in Sundance is using the processor architecture efficiently by calling the efficient BLAS routines for matrix-vector and matrix-matrix operations.

We assume that the resulting stiffness matrix and vector for the 7-th cell are the ones shown in Fig. 5.3. In the next step, we need to map the local DoFs of Cell 7 to the global DoFs, such that the contribution of this cell can be added accordingly. The global DoFs are given by the DoF map that will be presented in the next subsection.

[6] This case is a 2×2 matrix.

Once the global DoFs are available, as a last step for Cell 7, the **fill-in** process takes place. This assumes that the system matrix A and vector $\mathbf{b}$ were initiated with zero entries and during the assembly process, the contributions of the integrated cells and elements will be added to A and $\mathbf{b}$. This fill-in process is illustrated on the right side of Fig. 5.3, where only the marked entries in the vector $\mathbf{b}$ and matrix A will be affected. At the end of the assembly process, the resulting matrix is then $A = \sum_{i=1}^{7} A_i$ and the vector $\mathbf{b} = \sum_{i=1}^{7} \mathbf{b}_i$.

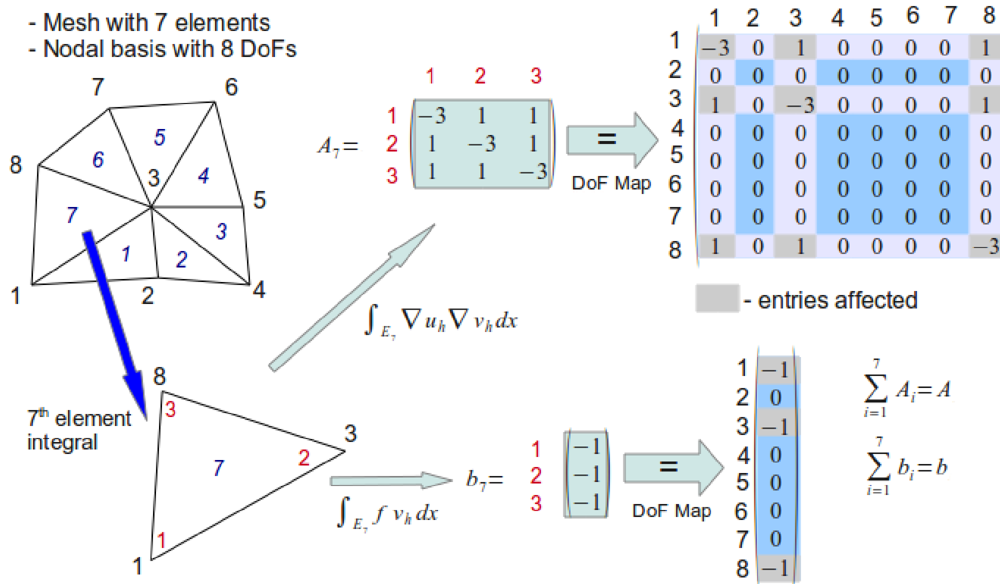

Figure 5.3.: Illustrates the matrix assembly process of a simple mesh (top left) with 7 cells. We use nodal basis and with the 7 (blue marked) cells and the problem results in 8 unknowns (global DoFs). The 7-th cell is picked out and two different weak forms are computed: the stiffness matrix $a_{i,j} = \int_{E_7} \nabla u_{i,h} \nabla v_{j,h} dx$ and the right-hand side $b_i = \int_{E_7} f v_{i,h} dx$, with $f \in \mathbb{R}$. For the case of simplicity, we assume the results of these integrals are the numbers in the 3×3 matrix A_7 and in the 3×1 vector b_7. The red numbers represent the local DoFs of the cell that do not correspond to the global DoFs. Mapping the local DoFs to the global DoFs is done by the DoF map. In the next stage, the fill-in process takes place, where with the use of the DoF map A_7 and b_7 is added to the global system matrix and vector.

Sundance offers not just integrals on cells, but integrals on lower dimensional elements. Code 3, from the previous section, contains such a declaration that is used for the Dirichlet BCs. Besides this type of BCs, Neumann BCs also require boundary integral implementations. Therefore, we illustrate in the following the assembly of lower-dimensional integrals. Fig. 5.4 represents the integration and assembly of one edge integral. We use the same mesh (left top) as in Fig. 5.3, but the blue numbering this time represents the

IDs of the edges. As one of the boundary edges, we pick the edge with ID 4 (Edge 4). The terms that are computed on Edge 4 represent the Dirichlet BC of the initial weak form example (5.3), with $u_0 = 1.0$. Further, it is assumed that Edge 4 is part of Γ, which the BC is imposed on. Similar to the 2D case, these integrals need to be computed only on the reference 1D cell and then transformed to a given edge.[7] The resulting element stiffness matrix and vector are showed in Fig. 5.4 and similar to the cell integration case, the element's results need to be mapped to the global system. This job is assigned to the DoF map, to deliver the global DoFs of a specified lower-dimensional element. With the global DoFs, the results can be further processed. In case of Neumann BC, the results go through the fill-in process and are added to the global systems, whereas the results of a Dirichlet BC have to be treated differently, as it will be shown in the second next subsection.

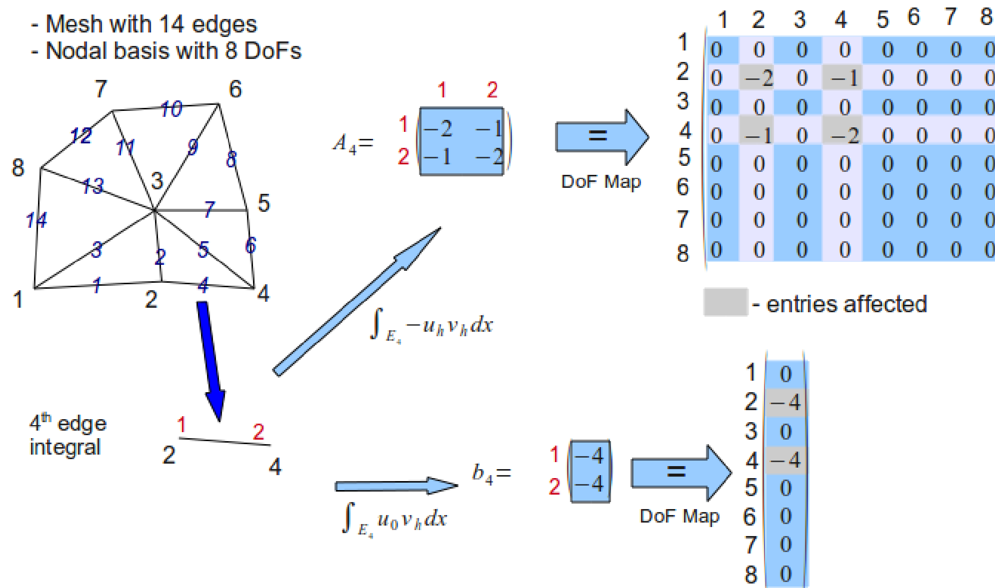

Figure 5.4.: Illustration of a lower dimensional element integration. We use the same mesh and nodal basis functions as in Fig. 5.3, but here the blue numbers represent the IDs of the edges (see mesh interface). As an example, we pick the 4-th edge and we compute a right-hand side $\int_{E_4} u_0 v_h dx$, with $u_0 \in \mathbb{R}$, and the mass matrix $\int_{E_4} -u_h v_h dx$. Similar to Fig. 5.3, the fill-in process is governed by the DoF map that provides the global DoFs corresponding to the local DoFs of edge with ID 4.

Degree of Freedom Map

Previous examples in Fig. 5.3 and Fig. 5.4 already highlighted in advance the necessity of mapping the element's local DoFs to the global DoFs. Similar to the mesh entities,

[7]The transformation in this case is simply a scalar multiplication.

the global DoFs have to be identified uniquely by a positive number that represents the index in the unknown vector. In contrast to the mesh's geometric elements, the DoFs do not have any dimension associated and are numbered from 1 to R.

The global numbering of the DoFs, as illustrated in Fig. 5.3, is the task of the DoF map that is also an internal Sundance object. Besides the number of mesh elements, there are key information, which determine the number of DoFs. The basis function type and order determine the number of local DoFs on an element and the number of unknown and test basis function can also vary in applications. Our previous example of the Poisson equation (5.1) has only one scalar unknown and one associated test function. However, in the case of Stokes and Navier-Stokes equations the number of unknown fields is $d + 1$, where d is the dimensionality of the problem.[8] One other relevant factor is the omnipresence[9] of the unknowns. In some applications not all the unknowns are defined on whole Ω, but just on a subset of it $\Pi \subset \Omega$ (e.g., Lagrange Multiplier approach in Chapter 4).

For these reasons by the global DoF numbering, the DoF map needs main parts of the problems formulation that consists of the mesh and of all the unknown and test basis function types with their orders and their domains (CellFilters) where they are defined on. With this information, the DoF map is able to assign to each DoFs a unique global number. In contrast to the mesh interface, presented earlier, even on distributed systems the DoFs are numbered globally, and there is no need to number the DoFs only on the local mesh. The local DoFs, as they were introduced in Chapter 2, represent the DoF numbers on one reference element.

Once the DoFs were numbered globally, the assembly process can be started, which process was illustrated in Fig. 5.4 and in Fig. 5.3, where the DoF map is required mainly at the *fill-in* step. The mapping from the local DoFs of one cell or facet to the global DoFs is done by the DoF map object and this functionality of the DoF map is illustrated in Fig. 5.5. During the assembly process, the mesh entities are identified by their dimensionality and their local IDs and these information are the input for the DoF map. The returned mapping contains the global DoFs associated to the requested mesh facet or cell. In the case of multiple unknown and test functions, all the DoFs of the specified mesh entity are returned. These DoFs specify the column and row index of the fill-in, determining which entries will be affected.

The target of the fill-in might be either the right-hand side vector or the system matrix, depending on the integration term. Terms of the form $\int_\Omega f\left(D^\alpha v_h\right)dx$ with $f \in L\left(\Omega\right)$ that contain only test functions and no unknown functions are called **one-form** integrals.[10] One-form terms are always assembled to the right-hand side vector. Terms with the general form $\int g\left(D^\alpha u_h\right)\left(D^\beta v_h\right)dx$, $g \in L^2\left(\Omega\right)$ are called **two-form** integrals, because

[8] For these equations, in order to fulfill the inf-sup condition (see Chapter 3) the velocities must be discretized with different finite elements than the pressure field.
[9] omnipresent = defined everywhere
[10] D^α represents the differential operator from Chapter 2.

they contain both a form of the test and unknown functions. Results of these terms are assembled in the system matrix.

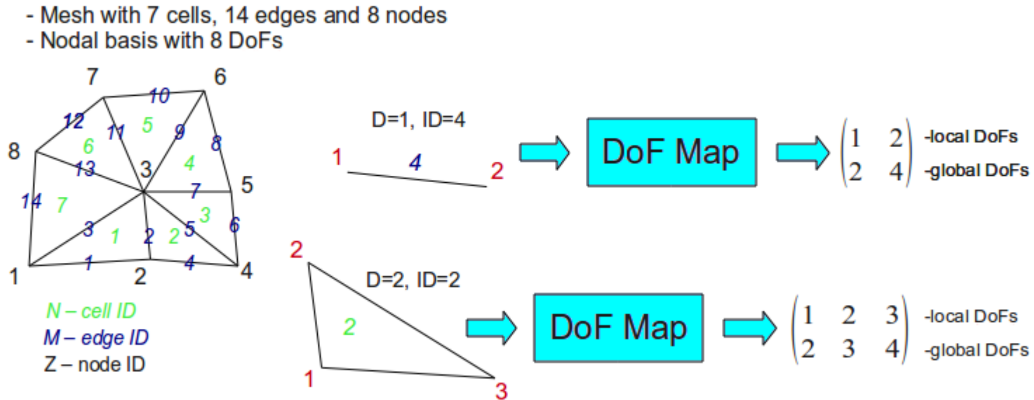

Figure 5.5.: Illustration of the DoF Map functionality. The simplex mesh (right) illustrates the IDs of the triangles, edges and nodes. The DoF map maps the local DoFs of a given mesh entity to the global DoFs (right). The mesh entities need to be specified by their mesh IDs and dimensions. In case of multiple unknown and test functions, all the associated global DoFs will be returned. The global DoFs coincide in this case with the edges' IDs.

The assembly of the one-terms requires only the DoFs of the test functions, whereas the two-forms require the unknown and test function's DoFs. However, in a given term these functions might be different, such that the column index does not coincide with the row index. Therefore, the convention is that the row indices are the unknown DoFs and the column indices are the test function's DoFs. Such terms are also called *mixed* elements, which are often used in the flow simulation.

We consider the example presented in Fig. 5.6, where in 1D a mesh given by a single element is shown. The integral over the element results in a rectangular 2×3 matrix, where the rows represent the local DoFs of the unknown function and the columns are the local DoFs of the test function. The DoF map then maps these local DoFs to the corresponding global DoFs, such that the row and column indices of the fill-in result as illustrated in Fig. 5.6.

By the fill-in process, the column indices differ from the row indices, not just when using different type or order basis, but also when one uses multiple scalar fields (e.g., Q_1Q_1 Elements). These cases are treated also according to the illustration in Fig. 5.6.

These functionalities of the DoF map are used not just during the assembly process, but also when the resulting unknown vector $\mathbf{x}$ needs to be mapped back to the mesh, for visualization or evaluation purposes.

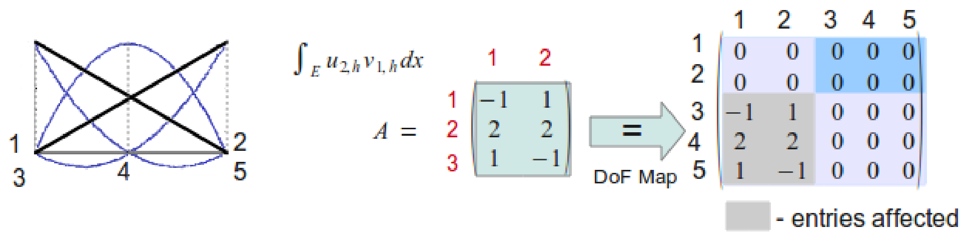

Figure 5.6.: This illustration shows a simple example of mixed elements in 1D (left). We consider only one mesh element in 1D with one linear (black) and quadratic (blue) basis. The global DoFs are numbered from 1 to 5 as shown on the left, DoFs 1 and 2 are assigned to the linear basis, whereas DoFs 3, 4, and 5 represent the quadratic basis. The assembly is illustrated for the mass matrix of $\int_E u_{2,h} v_{1,h} dx$, where $v_{1,h}$ is the linear test function and $u_{2,h}$ is the quadratic unknown function. The resulting matrix is assumed to be A. The Fill-in into the 5×5 system involves the DoF map.

Integral with Cell Filters and BC

In the problem formulation section, the cell filters were already mentioned, where they define the computational domain Ω. They also define the domains, where the test and unknown functions are defined and this information is further important to the DoF map. Cell filters also provide the stream of elements for the assembly process of the system matrix, where each element is assembled according to the presented approach. The cell filters are a general mechanism not just to define the problem, but also to use different quadrature methods for different categories of elements.

In the following, we enlist shortly how Dirichlet BC is imposed inside Sundance, using the cell filters and the $EssentialBC$ construct. Sundance uses the method described by Equation (2.19) in Chapter 2. This approach consist of marking first all unknowns, which are impacted by the cell filters of $EssentialBC$. For elements within the cell filter of $EssentialBC$, the global DoFs are collected, and these DoFs have to be treated separately. The corresponding rows of these DoFs in the matrix and in the right-hand side vector are set to zero. In the last step of imposing the Dirichlet BC, the element wise results of the weak form BC are filled-in into these rows of the system matrix and right-hand side vector.

By using this approach, the DoFs that are specified by a Dirichlet BC are not factored out to the right-hand side and are entries of the unknown vector. This approach involves the solving of a larger system, compared to the right-hand side factored approach that, especially in 3D, might decrease the size of the system significantly. On the other side, this approach of dealing with the BC has some advantages for optimization problems, in cases when the control variables are on these boundary and these DoFs remain part of the unknown vector.

Parallel Matrix Assembly

In the distributed memory case, once the DoF map is set up and each local DoF has a global DoF, the next step is to assemble the matrix. We want to point out here that in parallel cases, where the mesh is also distributed among the processors, the matrix assembly can be done in an unsynchronized parallel way. Thanks to the ghost cells, which are parts of the local mesh, all the row entries of a global DoF, owned by the local processor, can be computed locally without any additional communication with neighbor processors. This feature is crucial in order to have a good parallel scaling of the matrix assembly process on distributed memory systems.

Nonlinear case

In the problem formulation section, we presented the setup of a nonlinear problem and Sundance enables the declarations and solving of such nonlinear problems in the same abstract way such as linear problems. The only distinction in the problem declaration (see Code 5) is the definition of an initial value and a nonlinear solver of the problem.

However, in the solving mechanism, a nonlinear system needs to be solved. This solving mechanism for nonlinear problems within Sundance is described in a more detailed way in [62]. In the following, we highlight the key ideas and methods to deal with such problems.

Since fix-point methods are often not the method of choice, the first order derivatives of the problem functional are needed. To illustrate this, we consider the functional $F(u)$ that represents the nonlinear problem $F(u) = 0$. The input is the unknown function u defined on Ω, $u = \sum_{i=1}^{N} x_i \psi_i$. Using the gradient information at the actual solution u_k in a Newton type method speeds up the convergence significantly, and in many cases is even required to achieve convergence at all. With the definitions above the derivative can be stated as

$$\frac{\partial F}{\partial x_i} = \int \frac{\partial F}{\partial u} \frac{\partial u}{\partial x_i} = \int \frac{\partial F}{\partial u} \psi_i. \tag{5.5}$$

The only term in (5.5) that has to be treated symbolically is $\frac{\partial F}{\partial u}$. In Sundance, such terms are considered as the Fréchet derivative [62] of the functional F. This symbolic object does not contain any information about the discretization of the problem. Therefore, it can be computed by automated differentiation (AD), as it is described in [62].

During a nonlinear solving process the derivatives need to be evaluated, in each Newton step. Hence, the numerical evaluation of the symbolic objects such as $\frac{\partial F}{\partial u}$ need to be done in an efficient way. Sundance uses a symbolic graph representation that is later transformed to a simpler and efficient form. During computations, this graph remains static allowing fast evaluation of these symbolic objects.

This mechanism enables also the computation of spatial derivatives of the basis function that is commonly used in stiffness matrix's assembly. For more details on this symbolic representation and AD we refer to [62].

5.1.3. Solvers

In the linear case, once the matrix is assembled, the resulting linear system needs to be solved, whereas in the nonlinear case a nonlinear solver is required. Since Sundance is part of the numerical library Trilinos [43], it can access to all the linear and nonlinear Trilinos solvers, including various preconditioners. These solver packages are e.g., AztecOO, Belos and Amesos. In addition, Sundance can also access through interfaces third party solvers such as the sparse system solver SupreLUDist [57]. Using preconditioners with problem specific setup could further reduce the computation time. For this reason, Sundance has also access to Trilinos preconditioners e.g., IFPACK [77] and the algebraic multigrid preconditioner ML [34].

These solvers are selected via an XML configuration file that also contains the necessary parameter configurations for the solver (e.g., maximum number of iterations, tolerance). We illustrate this in Code 6, where a BiCGStab Belos solver is initiated from the *bicgstab.xml* file.

Code 6 Declaration of a linear solver. The *prob* variable was declared previously as a *Linearproblem* object in Code 4.

```
ParameterXMLFileReader reader("bicgstab.xml");
ParameterList solverParams = reader.getParameters();
LinearSolver<double> solver
  = LinearSolverBuilder::createSolver(solverParams);
Expr up = prob.solve(solver);
```

In the nonlinear case, Sundance can access the NOX & LOCA nonlinear solver package within Trilinos. The declaration of these solvers is similar to the linear ones and this is shown in Code 7. Solving a nonlinear problem is not always successful and it might depend on the fine tuning of the Newton solver. In case of failure, when $|F(u)| < \varepsilon$ could not be achieved, the *status* flag is set accordingly.

Simulations on distributed memory systems requires in Sundance solver libraries, which are also capable of solving (non)linear systems. The iterative solvers within Trilinos are capable of efficient distributed memory solving, whereas the direct solvers such as the ones is Amesos, do not fit well to parallel computations. The nonlinear solver NOX is also capable of efficient distributed memory simulations, if the chosen underlying linear solver for the linear step is an efficient one.

Code 7 Declaration of a nonlinear solver. The *prob* object was declared as a *Nonlinearproblem* object declared in Code 5.

```
ParameterXMLFileReader reader("nox-amesos.xml");
ParameterList noxParams = reader.getParameters();
NOXSolver solver(noxParams);
StatusType status = prob.solve(solver);
```

5.1.4. Visualization

Once the solution is computed in the last stage of the simulation pipeline the results might be further evaluated (e.g., computing errors) or visualized. Sundance does not include a visualization part, instead writes out the results in several standard formats, which can be further visualized by an external tool. The available standard formats are VTK, ExodusII, and Matlab.

Code 8 shows the exporting of a flow field into a VTK file. This example code exports all scalar fields and the velocity vector field. The output files can then be visualized by an external visualization software, such as Paraview [71].

Code 8 VTK visualization of a Stokes flow field. The solution is contained in the *soln* expression.

```
Expr soln = prob.solve(solver);
FieldWriter w = new VTKWriter("Stokes2D");
w.addMesh(mesh);
Expr expr_vector(List(soln[0],soln[1]));
w.addField("ux", new ExprFieldWrapper(soln[0]));
w.addField("uy", new ExprFieldWrapper(soln[1]));
w.addField("vel", new ExprFieldWrapper(expr_vector));
w.addField("p", new ExprFieldWrapper(soln[2]));
w.write();
```

In case of simulation on distributed memory systems, the visualizations (e.g., exporting to VTK files) needs also to be done in parallel. In this case, each process has access only to the local mesh and the results on it, and only the local results are plotted. Among others, the VTK format enables parallel plotting that is exploited within Sundance, such that each process plots its local results in separate files. External visualization tools such as Paraview [71] are capable of visualizing these local files as a global result on the global mesh.

5.2. Overview of Open-source FEM-based PDE Toolboxes

Until this point, the high-level descriptive language with C++ Sundance objects were introduced and we also highlighted the internal structure and assembly mechanism of the toolbox. Since Sundance is not the only existing FEM-based PDE toolbox, in the following, we give an overview of the existing FEM-based open-source PDE toolboxes and we also compare their features to Sundance. Due to the generality of the FEM approach and the natural need for code reuse, there are other research projects to establish a framework or toolbox for general FEM-based PDE solving.

In the following, it will be shown that Sundance has unique features compared to most PDE toolboxes. These features are, among others, the high-level problem description language in C++ and the capability to simulate problems efficiently in parallel on distributed systems.

FEniCS

The FEniCS[11] project was originally started as an implementation to evaluate weak forms [53] for a FEM approach, based on code generation. Later became a collection of several numerical packages [59] that enable the automated solution of PDEs by FEM. These included packages are *DOLFIN*, *FIAT*, *FErari*, *UFL*, and *Viper* that mainly enable the general problem formulation. The weak form is formulated in FEniCS with *UFL* description that is similar to the Sundance descriptive language. It uses high-level object based description in Python that can be used either in Python environment to compute and visualize the solution directly, or is compiled into a C++ code that can be used as simulation code. In the second case, the user has also to write a C++ code that includes and uses the generated C++ module. FEniCS also has AD capabilities, which enable the automated computation of derivatives that is necessary for nonlinear problem solving. On the mesh side, it has internal meshes and also interfaces to external mesh creation and partitioning libraries such as SCOTCH [73] and ParMETIS[12]. For the linear solvers, similar to Sundance, it accesses the external libraries PETSc [79], Trilinos [43], uBLAS, and MTL4. These features enable FEniCS the OpenMP and MPI simulation of a given problem. It is more important that in the distributed memory case no processor is required to hold the global mesh. Overall, we can summarize that FEniCS also has a high-level descriptive language that also enables the direct definition and solution of nonlinear problems. In contrast to Sundance, this descriptive language is based on code generation. For more details on this toolbox we refer to [59, 33].

[11] http://fenicsproject.org/

[12] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

deal.II

deal.II (Differential Equations Analysis Library)[13] was introduced more than twelve years ago in [8]. In contrast to the previously presented toolbox, deal.II has a lower level descriptive language, where this language is C++ and more user specification is required to define a PDE problem in the weak form. To illustrate what this detailed description implies, we consider the assembly loop of the system matrix in Code 9. In Sundance and FEniCS, this assembly loop is hidden from the user, whereas in deal.II this loop has to be written by the user. This rule also holds the weak form declaration that in deal.II needs to be declared explicitly by directly using the basis function declaration. On the other hand, this lower level description facilitates a direct access to the solving mechanism. For instance, in deal.II, the direct manipulation of the matrix entries and the sparsity pattern can be made rather easily. A similar operation in Sundance would require the access of the lower level Sundance or even Trilinos objects. deal.II is famous to support hp-refinement [9, 7] that requires sophisticated DoF handling. The feature to facilitate both h- and p- refinement is not common among the PDE toolboxes. Besides this, it also supports Discontinuous Galerkin (DG) approaches as well. From version 7.x, deal.II has capabilities for massively parallel simulations, as demonstrated in [6]. With the underlying *p4est* mesh [27] the authors of [6] compute problems with up to thousands of cores and with hundred million unknowns. Due to the absence of AD capabilities, deal.II is able to solve directly only linear problems. Therefore, nonlinear problems have to be linearized by the user. Similar to other FEM-based PDE toolboxes, it has interfaces to several solver libraries (e.g., PETSc, Trilinos), and relies on them for efficient solving.

libMesh

The libMesh[14] toolbox is largely developed by the CFDLab [54] at the University of Texas.[15] Regarding the level of the problem formulation language, which is in this case also C++, libMesh has a similar structure than deal.II. It also requires from the user to explicitly define the assembly loop as illustrated in Code 9. This lower level access to the toolbox objects facilitates special intervention for special cases that might be required for some applications. libMesh only allows for the direct solving of linear problems, and nonlinear problems need to be linearized by the user. In cases of distributed memory systems, libMesh is more restrictive than deal.II. It stores the global mesh on each processor and further it decomposes with external packages (e.g., ParMETIS[16]). The storage of the global mesh on each node poses a significant bottleneck. After this stage, the assembly and solving is done in parallel with external solver package (e.g., Trilinos,

[13]http://www.dealii.org/
[14]http://libmesh.sourceforge.net/index.php
[15]http://cfdlab.ae.utexas.edu
[16]http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

PETSc).

Code 9 libMesh example of the Poisson equation's ($\int_\Omega (\nabla u \nabla v + f_{xy}\, v)\, dx = 0$) assembly loop with the main loop over the elements. The first triple loop represents the quadrature of the matrix entries, whereas the second double loop assembles the right-hand side. A similar code structure is used in deal.II as well.

```
const std::vector<std::vector<Real> >& phi=fe->get_phi();
const std::vector<std::vector<RealGradient> >& dphi=fe->get_dphi();
for ( ; el != end_el; ++el) { ...
       for (unsigned int qp=0; qp<qrule.n_points(); qp++)
          for (unsigned int i=0; i<phi.size(); i++)
            for (unsigned int j=0; j<phi.size(); j++)
              Ke(i,j) += JxW[qp]*(dphi[i][qp]*dphi[j][qp]);
       for (unsigned int qp=0; qp<qrule.n_points(); qp++) { ...
          for (unsigned int i=0; i<phi.size(); i++)
              Fe(i) += JxW[qp]*fxy*phi[i][qp];
        ...
```

DUNE

DUNE (Distributed and Unified Numerics Environment)[17], represents a general framework for PDE problem solving that is not restricted to the FEM. It enables also the discretization and solution with finite volumes (FV) or with finite differences (FD) technique. DUNE has a mesh interface that facilitates distributed memory computations and also contains several mesh implementations (AlbertaGrid, ALUGrid, GeometryGrid, SGrid, and YaspGrid). The classical FEM approach enables usually only the leaf view of the mesh, but DUNE offers also a tree view of a given hierarchical mesh [12, 11]. This enables the implementation of various multigrid methods, which exploit the hierarchical structure of the mesh. DUNE offers two additional modules (DUNE-FEM[18] [29] and DUNE-PDELab[19]), which contain features for a FEM, FV and FD based solving approach. Since DUNE and DUNE-FEM offers a wide variety of features, the descriptive C++ language is even lower level than it was the case for deal.II and libMesh. For distributed memory systems DUNE offers not just parallel mesh implementations but also parallel matrix assembly and solving methods.

[17] http://www.dune-project.org
[18] http://dune.mathematik.uni-freiburg.de/
[19] http://www.dune-project.org/pdelab/index.html

6. Parallel Adaptive Cartesian Meshes in Sundance

In Chapter 5, we introduced the base line software architecture that was the status of Sundance before the methods and features described in this thesis have been implemented. Starting with this chapter, we introduce our developments to Sundance that enable a general IB method implementation solely based on a weak formulation, and also enable efficient simulation on distributed memory systems. This chapter presents the extensions of the Sundance PDE toolbox by rectangular elements and by an adaptive parallel Cartesian mesh in 2D and 3D, which is the first step towards IB methods. While adaptively refining a Cartesian mesh, hanging facets naturally arise, and these facets and their DoFs require a special treatment in order to ensure C^0-continuity at the cells' boundary. To tackle this issue in a user-transparent- and toolbox-manner, we developed the **pre-fill transformation** method that we introduced in [18]. This method required the extension of the mesh interface and implies an additional modular stage in the matrix assembly process [18]. We further present the current parallel and adaptive Cartesian mesh implementation and compare it to other alternative mesh implementations.

6.1. Quad and Brick Elements in Sundance

Sundance originally contained only simplex meshes, and the first step towards adaptive Cartesian mesh integration is the extension of the basic element classes with regular elements. The basis function of the elements is the classical Lagrangian polynomial, and the mathematical description of these elements was already given in Chapter 2 for 2D and 3D.

An element is basically a collection of DoFs, which are assigned to facets or cells on the reference element, and each DoFs has also an assigned basis function. An implementation of a given finite element would only require the declaration of these types of information. Sundance has a general interface to finite element implementations, therefore, the extension with these regular elements only implied the implementation of the following methods:

1.) *DoF location:* On the reference element, each DoF is associated either to the cell

or to one of its facets. The support of a basis function that is associated to a DoF might cover several elements, hence, it is important to recognize these DoFs during the global DoF numbering. Therefore, such DoFs have to be placed on the facet of a cell. This information is also crucial for facet integration, especially for imposing Dirichlet BCs.

2.) *DoF's basis function evaluation:* For a given point, specified in reference element coordinates, the value of each DoF's basis function is returned. Besides the value of the basis function, the spatial derivatives in all direction are also computed and returned. However, the computation of the derivative values is done automatically by automated differentiation (AD). The values of the basis function are mostly used for the various quadrature evaluations.

The methods above represent the modular interface to a general element interface in Sundance, and only these functions are used by other components of Sundance to interact with the finite elements. By implementing these two methods for rectangular elements, it enables already the computations on regular Cartesian meshes. However, for adaptive Cartesian meshes with hanging DoFs, this interface needs to be extended as well.

6.2. The Pre-fill Element Transformation for Hanging Degrees of Freedom

In Chapter 4, we introduced the Cartesian mesh structure. We also showed that the most suitable and efficient data structure for such a mesh is a tree. During refinement of a Cartesian mesh, hanging facets arise naturally. A hanging node is a hanging facet of dimension zero. The problem that arises is to ensure C^0 continuity between two cells, where the intersection of these two cells is a hanging facet (see Def. 5.1.3 in Chapter 5). This implies applying restrictions to the DoFs that are associated to such hanging facets. Our aim is to develop a general method to deal with hanging DoFs, such that no user interaction is required, while we preserve the actual software structure of Sundance. In this section, we present our approach to deal with hanging DoFs in a general and user-transparent approach.

Next, we illustrate the problem of hanging facets and the associated DoFs in Fig. 6.1. Generally, we restrict ourselves to the hanging facet issue on a **1-irregular** mesh[1], and we consider only the leaf view of the mesh as it was described in Chapter 4. This leaf view prohibits any spatial overlapping of the cells, and such a scenario is presented in our concrete example. The mesh in Fig. 6.1 is given by three cells, marked with blue numbers and the underlying basis function is assumed to be bilinear. Assuming that all local DoFs of the cell are also global DoFs, then the mesh would have 8 global DoFs. With 8

[1] neighboring cells have at most a level difference of one, see Chapter 4

global DoFs, the interface between cell 1 and cells 2 and 3 would be C^0-discontinuous. The potential discontinuity is shown in Fig. 6.1, where the basis functions on the cell boundary are illustrated. In order to eliminate this discontinuity, the red basis function should be restricted to the two marked global DoFs. The contributions of these global DoFs (2 and 6) are marked with the green line that is the sum of the basis functions marked with black (DoF 2) and magenta (DoF 6) colors. This example has a bisection refinement, and this means that the hanging DoF should be restricted to the value of $0.5x_2 + 0.5x_6$, where x_2 and x_6 represent the two global DoFs. We denote our hanging node's DoF with x_{HN}(red basis in Fig. 6.1), and the condition for C^0-continuity must hold as $x_{HN} = 0.5x_2 + 0.5x_6$.

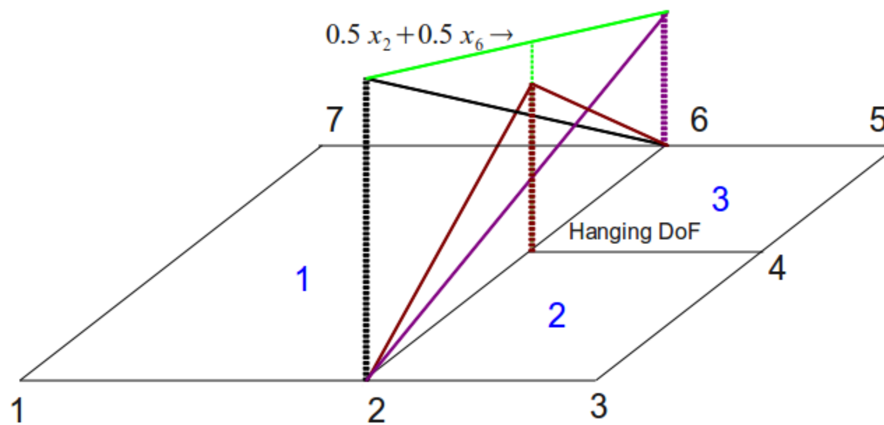

Figure 6.1.: Illustration of the potential discontinuity on the cells' boundary. The red DoF should be restricted to the global DoFs, such that it has the value of $0.5x_2 + 0.5x_6$. This sum is represented by the green line.

Generally speaking, one can state that DoFs owned by a hanging facet have to be treated in a particular manner to ensure C^0-continuity of the discrete solution. More concretely, DoFs at hanging facets are no real DoFs, but their values are determined by adjacent DoFs at non-hanging facets. The general constraint form for a local DoF x_i at a hanging facet can be formulated as

$$x_i = \sum_{k \in H_i} a_{i,k} x_k + b_i, \tag{6.1}$$

where H_i is the set containing neighboring non-hanging DoFs. The coefficients $a_{i,k}$ are the constraint coefficients and in most cases $b_i = 0$.

At this point, the question arises, how to compute in general case the coefficients $a_{i,k}$? We restrict ourselves only to Lagrangian basis functions with homogeneous order. The idea to compute the coefficient for a given order p is just simply evaluate the non-hanging DoFs' basis functions in H_i at the position of the hanging DoF. By evaluating the basis functions in Fig. 6.1, the resulting coefficients are 0.5 and 0.5, as it was already shown. This idea can be further extended to general nodal basis functions and to hierarchical

functions as well. From these considerations, it results that the determination of coefficients $a_{i,k}$ is assigned to the element as a new functionality. Therefore, the presented element interface is extended with one additional function.

1.) *Constraints for a hanging DoF:* For a specified hanging DoF i of a facet, it returns the set H_i containing only local DoFs of the parent cell and the coefficients $a_{i,k}$. However, the determination of the global DoFs needs further processing, since in this interface H_i is only given in local sense. Further, the refinement type (bisection or trisection), the hanging facet dimensionality, and its index in the parent cell need also be specified in this function.

This additional interface function plays an important role in the extension of the mesh interface that is described later on. In the following sections, we will come back to this functionality.

Imposing these constraints in a general toolbox-manner has been the subject of research within PDE toolboxes. One approach is described in [7, 6], where the constraints are stored and applied in their original form as in (6.1). This implies that during the **fill-in** process, if the targeted local DoF is hanging, the contribution of this DoF is distributed on columns and lines of the matrix according to the constraint (6.1). Treating each hanging DoF individually has also its advantage. In the example of Fig. 6.1, treating each hanging DoF individually implies the storage of the single hanging DoF, even though there are at least two cells impacted. However, we have a different view of this problem, instead of treating each hanging DoF individually, we look to the problem from the cell point of view.

We consider the cell view of the example from Fig. 6.1 in Fig. 6.2, and we illustrate the global and local DoFs of the three cells. In cell 1, there is no need for DoF constraints, but in cells 2 and 3, the local DoFs are constrained in the following way: $\hat{x}_3 = 0.5x_2 + 0.5x_6$ in cell 2 and $\hat{x}_1 = 0.5x_2 + 0.5x_6$ in cell 3, where $\hat{x}$ represents the local DoFs. In Fig. 6.2, we illustrate the local and global DoFs of the three cells from

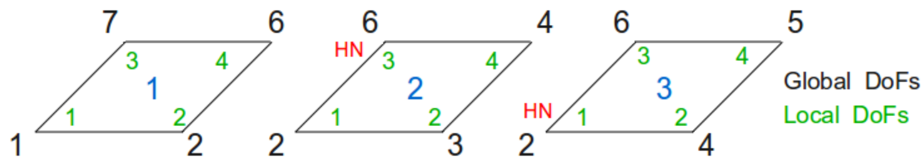

Figure 6.2.: Global and local DoF numbering of the bilinear quad element of Fig. 6.1 (similar to Fig. 2.3 in Chapter 2). The DoFs are only owned by the nodes of the cell. The numbers outside the cells represent the global DoFs and the green numbers inside the cells represent the local DoFs number.

our example. The local DoFs are denoted by green numbers according to the numbering convention of the quad elements, whereas the global DoFs are denoted by black numbers.

The single hanging DoF is marked by HN. It is important to note that each cell must have the same number of associated global DoFs. In the concrete example, cells 2 and 3 have also 4 associated global DoFs as cell 1. This relation also holds for different basis orders and refinement strategies, and this is a key property for the following sections.

The first key point of the cell view (and also of the facet view) is that a cell, even though it has hanging DoFs, always has a list of associated global DoFs. The next step is to find these global DoFs in a unique way, and for this information, additional mesh queries are required. The involved global DoFs in the constraints of (6.1) are not always adjacent to the cell. In such cases, we need the hierarchical mesh geometry information to determine the corresponding non-hanging DoF to each hanging local DoF.

By keeping the **1-irregularity**, it is assured that, if a cell's facet is hanging, the parent cell's facets with the same facet index can not be hanging. This observation is illustrated in Fig. 6.3 with trisection refinement strategy, where the parent cell is refined. We assume that the lower neighbor of the parent cell is not refined, creating hanging DoFs on that lower edge. We consider the child cell illustrated in Fig. 6.3, where the 0 and 1 local nodes are hanging nodes. By mapping this child cell back to its parent cell, we notice that these hanging nodes can be mapped to the parents cell's 0 and 1 local nodes (as the two arrows show in Fig. 6.3). These nodes of the parent cell, while having the 1-irregularity, can not be hanging. Even though the parent cell itself is not visible to the toolbox components, its facet, which is accessed by one child cell with hanging facets, is visible to the toolbox components as it is at the same time a facet of the neighboring leaf cell. In Fig. 6.3, this observation holds for the lower edge of the parent and child cell and not just for the nodes. It is also true for the general facet case and for bisection refinement. This is an important point and we summarize it for the general case in the following lemma.

Lemma 6.2.1 *While keeping the 1-irregularity, if a cell's facet is a hanging one, the facet of its parent cell with the same facet index must be non-hanging. At the same time, this facet is also a facet in a leaf cell, such that it is visible to the leaf view of the mesh.*

Proof: Directly results from the hanging facet definition and from the definition of the 1-irregular mesh, such that the level difference of neighboring cells is at most one. □

With this lemma, one can specify a list of all involved global DoFs for a given cell. In case of cells with no hanging facets, this list is assembled by considering all possible facets of the given cell. By listing the global DoFs owned by the facets, and by adding the DoFs owned by the cell itself, this list is created. We denote this list with T_E, where E is the cell index. For cell 1 in Fig. 6.2 and Fig. 6.1, this list is $T_1 = \{1, 2, 7, 6\}$. For each cell, which has a hanging facet, each non-hanging facet contributes its DoF (if any) in the usual way. Since the hanging facets do not own global DoFs, the respective DoF numbers are replaced by the facet's DoF numbers of the parent cell. For the cell 2, this list is $T_2 = \{2, 3, 6, 4\}$ and for the last cell, this list is $T_3 = \{2, 4, 6, 5\}$. We already

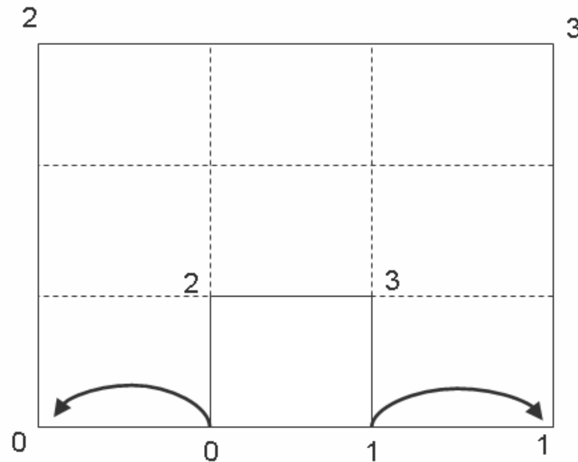

Figure 6.3.: Illustration of the hanging nodes, resulting from a trisection refinement. These hanging facets and the corresponding hanging DoFs can be mapped to the parent cell's facets and global DoFs, since with a 1-irregular mesh, they can not be hanging. The two arrows show this mapping of the hanging nodes to facets of the parent cell that are not hanging.

specified that for a given basis function, this list of global DoF numbers has the same length, regardless of the number of hanging facets in the cell.

Once these lists for all cells are created, the next step is to store the constraints from local DoF to global DoF defined by (6.1). Since the number of local DoFs must be equal to the number of entries in each of the global DoF list, a square matrix is well suited for this storage. One line of the square matrix represents one local DoF, whereas a given column belongs to one global DoF. For each hanging DoF, the respective line of the matrix contains the entries $a_{i,k}$ for all impacted k, and zero for other columns. For non-hanging DoFs, the given matrix line is the identity line:

$$M(i,k) = \begin{cases} a_{i,k} & \text{if i-th local DoF is hanging and } k \in H_i \\ 0 & \text{if i-th local DoF is hanging and } k \notin H_i \\ 1 & \text{if i-th local DoF is non-hanging and } i = k \\ 0 & \text{else} \end{cases} . \qquad (6.2)$$

We denote the resulting matrix with M and we call it transformation matrix. H_i is the set of all global DoFs, which are required to compute the value of a hanging DoF i. We illustrate the transformation concept by considering again the example in Fig. 6.1, with B_1 denoting the bilinear basis functions. The first cell with index 1 does not need a transformation (the transformation matrix M_{1,B_1} would be the identity), but the

transformation matrices for the other two cells have the form

$$M_{2,B_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad , \quad M_{3,B_1} = \begin{pmatrix} 0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

It is obvious that the structure and the entries of the transformation matrix depend only on two factors. The first one is the cell's position, e.g., the configuration of hanging facets and the second one is the basis function that we denote with B. For this reason, we also denote the transformation matrix belonging to a cell E with the given basis function B as $M_{E,B}$.

In real applications, the number of different basis functions is limited (the order p is fixed). For instance, in the case of Navier-Stokes equations, due to the inf-sup criterion, there are two different types of basis functions. However, in most applications, using one type of basis functions is usually sufficient for the FEM discretization. Once a basis function B is given, we note that the number of possible matrices $M_{E,B}, E \in I_N$ has a low upper bound, which does not depend on the number of cells in the mesh. This upper bound is given by the combination of the hanging and non-hanging status of the cell's facets. In 2D with bisection refinement, the number of possible cases, when one cell is holding at least one hanging facet, is only 8 (4 cases with two hanging nodes plus 4 cases with one hanging node). For this reason, it is more reasonable to not store a matrix for each cell E, but we only store a positive number for cells that have hanging facets. This number represents the respective index in the set of all possible transformation matrices for the specified basis B. This way, the amount of data that needs to be stored cellwise is limited to one integer. Since the number of basis functions is limited, the total storage required of the transformation matrices is also limited to a constant factor, and cellwise to only a couple of integers.[2]

For the example in Fig. 6.1, with a two-dimensional mesh, with bisection refinement, and with eight possible forms of the transformation matrix M, it results in a total storage requirement of only 128 (8×16) doubles (1kB). With this amount of data, we stored all the possible combination of constraints for hanging facets in a cell.

The Pre-fill Element Transformation

At this stage, we summarize the above stated observations and methods in our new developed approach that is based on the cell view of the problem, and we call it 'Pre-Fill Element Transformation' [19]. The main input for the transformation, to enforce the constraints specified in (6.1), are the following information:

[2]Representing the indices

1.) *List of involved global DoFs:* For a basis function[3] and for a given cell E, this is represented by the list T_E.

2.) *Storage of the constraints:* These constraints are stored and represented by the transformation matrix $M_{E,B}$, for a given basis function type B and cell E.

The matrix $M_{E,B}$ with the additional list of the global DoFs T_E provides enough information to perform the Pre-fill Element Transformation for a given cell E with at least one hanging DoF. Through this transformation, the constraints are applied, and this takes place during the assembly of the global system matrix. The Pre-fill Element Transformation process transforms the unknowns, before the fill-in process of the local stiffness or mass matrix into the global matrix takes place, such that the fill-in operation will add the correct results to the global system matrix (see Fig. 6.4). Mathematically, this transformation only implies additional matrix-matrix or matrix-vector multiplications. If we consider the matrix assembly then through this multiplication, each global DoF should get the correct contribution from the local cell, according to (6.1). The transformation matrix $M_{E,B}$ is introduced in such a form that a multiplication of the local element matrix or vector achieves just that. This means that before the fill-in of the local element's result only one optional matrix-matrix or matrix-vector multiplication takes place that is the Pre-fill Element Transformation.

In terms of software modularity, this approach proves to be also efficient. It allows the use of the same reference element quadrature components[4] for all elements. In the pipeline of the matrix assembling, the transformation only adds an additional and optional stage as shown in Fig. 6.4.

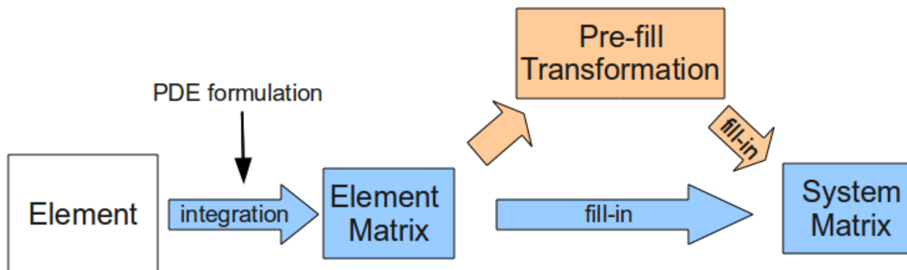

Figure 6.4.: The process of assembling the global system matrix. The Pre-fill Element Transformation is an optional stage in this process, and is used only for cells that have hanging DoFs. In the case of cells (or even facets) with hanging DoFs, the constraints are enforced with this transformation.

[3]This is not the basis function type but specific for the concrete basic function.
[4]as software components

In the first stage of the simulation, we consider the matrix assembly process, and for a cell E, we write the necessary transformations as

$$\begin{aligned} A_E^{new} &= M_{E,BT}^T \, A_E^{old} \, M_{E,BU} \\ b_E^{new} &= M_{E,BT}^T \, b_E^{old}, \end{aligned} \tag{6.3}$$

where BU represents the basis for the unknown function, and BT is the basis of the test function. A_E^{old} and b_E^{old} are the local (element) matrix and the local right-hand side, respectively, resulting from the classical element integration. We call the transformation that is required for the system matrix assembling, and maps the local element's DoFs to global DoFs, as **gather operation**.

However, it is also required to have an inverse mapping, from the global DoFs to the element's local DoFs. This is needed for non-linear problems, where the non-linear operator needs to be evaluated. It is also needed for visualization and different integral evaluations (e.g., error norm calculations) with the computed solution function. To compute the values of the local DoFs u_E^{local} from the values of the global DoFs u_E^{global}, the so-called **scatter operation** is applied:

$$u_E^{local} = M_{E,BU} \, u_E^{global}, \tag{6.4}$$

where the vector u_E^{global} has the values of the global DoFs specified by T_E, the global DoF list of cell E. Similar to the gather operation, the scatter operation is integrated modularly into the component of the element evaluation that is used for visualization and solution evaluation purposes.

Equations (6.3) and (6.4) represent the general case when the unknown and test basis functions BU and BT might be different. Therefore, both the test and unknown basis transformation matrices are required to make the correct transformation, according to the corresponding basis functions.

Until this point, we presented the Pre-fill Element Transformation as a numerical method to deal with hanging DoFs. However, on the software level, as it was already highlighted, this implies several extensions in the existing Sundance structure (e.g., the optional transformation stage). In the following sections, we highlight only the interface changes that impacted the mesh and the DoF map, and later, we also present the current adaptive parallel Cartesian mesh implementation.

6.3. Sundance Mesh Interface Extensions

Lemma 6.2.1 defines the key property that allows the determination of the global DoFs in the T_E list. This lemma requires access to the parent cell's facet that by definition is visible in the leaf view of the 1-irregular adaptive Cartesian mesh. To implement

this access to the mesh, additional functions need to be added to the Sundance mesh interface. In addition, the hanging facets of the mesh need to be identified also by the mesh interface. The extension to the existing Sundance mesh interface is represented by the following functions:

1.) *is facet hanging*: Specifies for a given dimension d and an facet ID[5] if this facet is a hanging one or not (according to Def. 5.1.3 from Chapter 5).

2.) *parent cell's facet*: If a facet is hanging, then the parent cell's facet is required to identify the global DoFs relevant for the value at the local hanging DoF. The parent cell's facet IDs are returned, which are visible in the leaf view and are used for hanging DoF handling (see Fig. 6.3).

3.) *cell's index in parent*: For a given refinement strategy, a parent cell will have a constant number of children, once this cell is refined. To compute the constraints coefficients, the index of the cell in the list of children of the parent cell is necessary. This index provides additional information to the element class about the position of the cell inside the parent cell and about the local IDs of hanging facets.

These functions need to be implemented only for the adaptive Cartesian meshes. For the existing unstructured meshes, these functions can have a trivial implementation. The DoF map is the entity that maps the global DoFs to the mesh entities. This component is extended for the adaptive Cartesian mesh, and it will detect automatically if the simulation is started on such an 1-irregular mesh. Therefore, during the simulation with regular unstructured meshes, these functions will not be called, and the trivial or no implementation in the unstructured case does not pose a problem.

6.4. Degree of Freedom Map Extensions for Hanging DoFs

The presented Pre-fill Element Transformation implies additional functionalities not just for the mesh interface but also for the DoF map. This transformation requires the list of the global DoFs T_E and the transformation matrix $M_{E,B}$ for one given element E. Both types of information fit best to the DoF map, as a modular component of Sundance. Therefore, the list T_E and the transformation matrix $M_{E,B}$ will be stored here. As described in the previous section, DoFs have global (mesh-wide) and local (valid in a cell) numbers. Once a local DoF is on a hanging facet, it needs to be constraint. This DoF is no longer a global DoF, hence, no global DoF number is assigned by the DoF map. The local DoF number is the index of the DoF inside an element and exists for all hanging or non-hanging DoFs. These local DoFs are then required to be mapped to global DoFs.

[5] facet index

In case of hanging DoFs, this mapping is done by the list T_E. Even though we defined this list T_E explicitly, this list can be computed on the fly, when the presented (see Chapter 5) DoF map interface function is called. The list of global DoFs can be delivered not just for a cell, but also for facets that hold or are themselves hanging mesh entities. The basis function B also plays an important role in the determination of this global DoF list. In cases of multiple unknown and test functions for one given cell, all the hanging DoFs associated to one hanging facet have to be treated accordingly. This way, we extended the existing DoF map within Sundance to store the local DoF to global DoF relations, even for hanging DoFs (with the T_E list).

The second component of the Pre-fill Transformation that is assigned to the DoF map is the transformation matrix $M_{E,B}$. These transformation matrices are generated and stored within this DoF map. We already pointed out that the number of transformation matrices for a given basis B is limited by a low upper bound. Since the number of different basis function types is also limited, the total storage requirement for these matrices is constant, and not depending on the number of mesh cells. The storage requirement of one transformation matrix is directly proportional to the number of DoFs per finite element. Assuming that there are D DoFs on one element and the matrix is stored as a dense matrix, D^2 double values need to be stored. In a three-dimensional example with second order basis functions, the required size is $D = 81$ and $D^2 = 6561$, and already one such matrix needs 52kB memory, where most entries are zeros. The chosen strategy to store the transformation matrix also defines the matrix multiplication algorithm of the dense[6] matrix A_E^{old} and the vectors b_E^{old} and u_E^{global}. In order to store and apply the transformations, we had mainly two different options:

- *Dense Storage*. This implies the full storage of the matrix, where most entries of $M_{E,B}$ are likely to be zero. On the other hand, it allows the usage of efficient BLAS2 and BLAS3 routines for the multiplication.
- *Sparse Storage* Alternatively, one can reduce the memory requirements by choosing a matrix compression scheme (e.g., CRS). This would save storage and also unnecessary multiplication operations (multiplications by zero), but would also require a special multiplication algorithm of a densely stored matrix with such a sparsely stored matrix.

In our implementation of the transformation matrix storage in the DoF map, we choose the first variant, and rely on the efficiency of the BLAS routines for matrix-matrix and matrix-vector multiplications, to outweigh the unnecessary multiplication operations. The number of cells that require Pre-fill Transformation is in general considerably lower than the total number of cells within the mesh. We already showed that the concrete number of transformation matrices that need to be stored is limited and is independent from the number of cells in the mesh. Therefore, we can conclude at the end of this section that the Pre-fill transformation does not pose any significant computational

[6] we used non-orthogonal basis function

overhead either in terms of additional operations or in terms of memory consumption.

6.5. Parallel Adaptive Cartesian Mesh Implementations in Sundance

After we presented the extension of the mesh interface, the next step is to describe the implementational ideas of the developed parallel adaptive Cartesian mesh within Sundance. This might sound simple, but practically the implementation of such a mesh alone could be subject of a research project, as it was the case in p4est [27], Peano [92] and FEniCS [58]. Therefore, we restricted ourselves to a simpler implementation that has its limitation in massive parallel simulations and in accurate load balancing. These limitations will be presented in more detail later in this section. At the same time, we want to underline here that the extended mesh interface and the Sundance components (e.g., DoF map, external solvers) generally support such massive parallel simulations, and do not pose any conceptual bottleneck for such simulations. To develop a more efficient and sophisticated parallel adaptive Cartesian mesh within Sundance, and to test massive parallel runs with Sundance could be subject of future research.

The actual Cartesian mesh implementation has the same underlying concept that p4est [27] has, which is namely to generate first one underlying regular mesh that is subject to further refinement and coarsening, while keeping the 1-irregularity. In contrast to the p4est mesh, we employ trisection based refinement that is characteristic to the Peano curve. The trisection implies more implementational overhead especially in the 3D case. In terms of transformation matrices, needed for the Pre-fill Element Transformation, the trisection also increases the number of possible cases. On the other hand, it allows a more aggressive refinement in the locality of interested area.

Even though Cartesian meshes allow for efficient tree storage, Sundance's mesh interface considers the mesh as a 'database', where one query is based on the mesh entities' IDs. Therefore, an implementation of the mesh based only on a tree storage that can efficiently answer the queries, turns out to be challenging.

Although the mesh interface allows random access, the mesh entities are accessed iteratively by the other components of the toolbox. This feature can potentially facilitate one iterator based mesh implementation as well. The concept of mesh iterators is widely used within PDE toolboxes (e.g., deal.II [7], FEniCS [58] and DUNE [29]), and this is also used within Sundance. However, during one iteration of a given dimensional mesh entity, the facets and the co-facets of the actual mesh entity can also be subject of the queries.[7] For this reason, a tree and iteration based implementation poses a significant challenge. In the following section, we present such a tree and iteration based mesh

[7] This would imply access to neighboring cells.

integration into Sundance, and we show also their benefits and drawbacks.

One obvious way to answer the queries efficiently, is to linearize the tree, and to store all necessary facet and co-facet information regarding one mesh entity. This way, most of the mesh accesses consist of only storage access without additional computations that allows random access to the mesh. To illustrate this concept, we consider one quad facet in 3D. For this all the node and edge facets need to be stored, and the maximal co-facets of this quad is required as well. According to the mesh interface, this also implies the storage of the nodes' position, similar to unstructured meshes. In contrast to the unstructured mesh, the Cartesian mesh has a well defined structure, therefore, it allows a more efficient data structure even for the storage of the linearized adaptive Cartesian mesh. This will be demonstrated in a later section where we compare the memory requirements of different meshes.

Initially a regular Cartesian mesh is generated with the defined resolution in 2D or 3D. The traversal and the numbering of the entities of our adaptive Cartesian mesh are based on the Z-curve, where only the coarsest cells are traversed. Within a tree of a coarsest cell, the children are traversed with a breath first algorithm. This traversal is illustrated in Fig. 6.5.

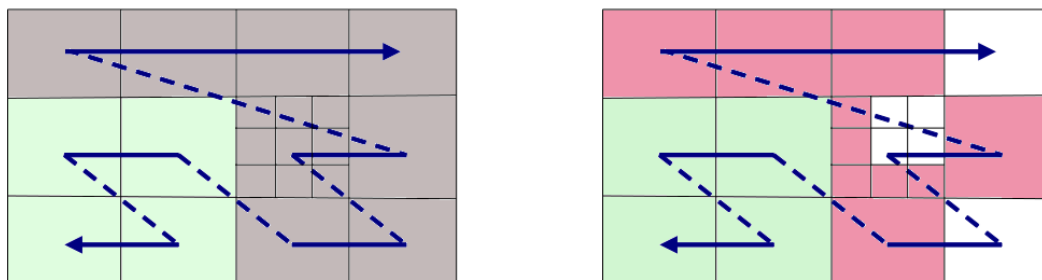

Figure 6.5.: A simple example of a parallel adaptive Cartesian mesh that is partitioned into two domains. On the left, it illustrates the partition of the domain based on the Z-curve. On the right, it shows the mesh partition of processor 0, with the ghost cells (marked with pink). It is assumed that the facets on the interprocess boundary also belong to processor 0. One can observe that not just the interprocess boundary cells are added as ghost cells, but also those that are required for the hanging local DoFs, and also for the Pre-Fill Element Transformation.

In the parallel case, the required domain decomposition is based on this Z-curve traversal. In the following, we highlight the decomposition of our Cartesian mesh. The desired outcome of this process is a load-balanced decomposition of the mesh, and one of the most efficient methods to partition a Cartesian mesh are based on various space-filling curves. This was already demonstrated by the p4est mesh [27] that is based on the Z-curve. In the case of the Peano mesh [92] with trisection refinement, the decomposition

is based on the Peano space-filling curve.

Before the partitioning starts, similar to p4est [27], a regular coarse mesh is created. This regular mesh is then refined according to a call-back function that can be either an error indicator or a predefined function, while during refinement the 1-irregularity is kept. The outcome of the refinement is an adaptive Cartesian mesh that needs to be partitioned into P equal partitions, where P is the number of processors in the parallel computation. The current mesh implementation, as we mentioned earlier, is under development and actually, it partitions only the initial regular mesh that is formed by the coarsest cells. This Z-curve based partitioning is illustrated in Fig. 6.5, where the coarsest cells, with their complete tree, are assigned to one processor. This way, the cells on the finer refinement levels have the same owner processor as the coarsest parent cell. Cells, which are created by the refinement, are partitioned with the coarsest parent cell. Before partitioning, a load indicator has to be assigned to each coarse cell that in our case is the total number of child cells within the cell. After the total load of the mesh is estimated, the coarse cells are distributed along the Z-curve in a "greedy" manner.

In addition to the mesh partitioning, ghost cells are necessary for the correct computation of matrix entries at the boundary of a process domain. In the case of adaptive Cartesian meshes, these ghost cells are not only those cells with a facet on the processes subdomain boundary. As depicted in Fig. 6.5 (left) not only direct neighbor cells of the mesh partition need to be added, but also cells that contain DoFs and have a contribution to the DoFs lying on the interprocess boundary. In the case of hanging facets, this might also include parent cells, required for the Pre-fill Transformation. Such a decomposition for a more complex scenario is shown in Fig. 6.6.

This implementation of the partitioning of the adaptive Cartesian mesh is rather simple and the determination of the ghost cells (including the additional ghost cells if necessary) is also done in a straight-forward manner. However, the current Cartesian mesh implementation in 2D and 3D requires the global storage of the mesh, even though after the partitioning only the local mesh is further used. This poses a significant storage and computational overhead that could be addressed in the future. Besides, the partitioning solely based on the coarsest cells also poses a significant bottleneck for deeply refined meshes. Once a coarse cell contains a large number of child cells, it will become a bottleneck, since the load can not be split between processors. This issue should be also addressed in future developments.

In the following, we enlist a small example of code, how the developed adaptive Cartesian mesh can be accessed within Sundance. Code 10 shows the declaration of the adaptive Cartesian mesh within the Sundance's descriptive language, where a 2D adaptive Cartesian mesh is created. In this case, the adaptivity is solely based on a callback function that can decide whether a cell should be further refined or not, given the cell's position and refinement level. This actual mesh implementation has also the option to deactivate a set of cells, defined by the class *MeshDomain* in Code 10. At the beginning, the mesh is initiated on a regular domain, but with this deactivation only the specified

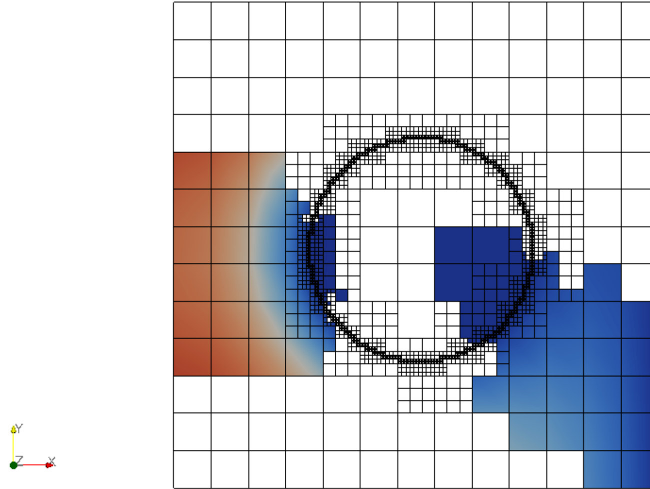

Figure 6.6.: Domain decomposition of an adaptively refined Cartesian mesh into eight equal domains. The illustration shows the load-balanced partitions of the second and fifth processors.

portion of this domain becomes visible for the simulation. One of the main advantages of this Cartesian mesh is its usability in the parallel case, where for any given processor numbers, the Sundance code remains the same. Since the mesh partitioning is done internally in the background, the same compiled code can be started with different numbers of processors on distributed memory systems.

6.5.1. Mesh Storage and Runtime Comparison

In the following, we compare various mesh implementations within the frame of Sundance. The scope of this comparison is to test the performance of the previously presented adaptive Cartesian mesh implementation. We measure the performance of the mesh indirectly, by comparing the total runtime and storage requirement of simple serial PDE computations.

Before describing the scenario and enlisting the results, we shortly describe the two other mesh types that we tested. The first mesh type is the unstructured mesh with simplex elements (called simplex mesh) that was already mentioned in the previous chapter. The second type of mesh that we tested is a regular mesh prototype and is a Sundance integration of the Peano mesh [92]. The Peano mesh uses a tree based storage and a stack-based traversal, and allows mesh access only by an *adapter concept* [92]. This further implies that only an iterator based traversal is enabled and even for the regular case turned out to be technically challenging. Since the parallel and the adaptive

Code 10 Creation of a 2D Cartesian adaptive mesh. The adaptivity is defined by the callback function of *MeshRefEst*. This callback function gets the actual position of the cell, including the size of the cell, and the actual refinement level *cellLevel*. This callback mechanism can also be used for other types of refinement. In addition, the user has the option to use only a subpart of the mesh. The *MeshDomain* class defines this particular domain. Cells, where the callback of *MeshDomain* returns true, will be inactive cells, and will not be visible to the other component of the toolbox.

```
REFINE_MESH_ESTIMATE(MeshRefEst,{
if (((cellPos[0]>0.5)&&(cellPos[1]<1.0)&&(cellLevel<2)))
   return 1; else return 0; } , {return 1;} )
MESH_DOMAIN( MeshDomain , {return 1;})
...
RefinementClass refCl = new MeshRefEst();
MeshDomainDef meshDom = new MeshDomain();
MeshType meshType =MeshRefEst new HNMeshType2D();
MeshSource mesher=new HNMesher2D(0,0,1,1,81,81,meshType,refCl,meshDom);
Mesh mesh = mesher.getMesh();
```

case represented further technical challenges and workarounds, we abandoned the full integration of the Peano mesh. The iterator based approach implies for a complete PDE simulation several mesh traversals, in order to answer the queries that might be costly, in comparison with direct memory access. On the other hand, this approach has the prospect of being much more memory efficient, since most of the required information (e.g., node positions) are computed on the fly.

To investigate these three different variants, we chose one of the simples PDE, the Poisson equation in 2D and 3D. In these simple computations, the mesh is used in the DoF map building and in the matrix assembly and visualization, hence, the underlying mesh implementation plays an important role. In 2D, we tested two configurations that fit to the regular Peano mesh on the unit square [8]. In order to minimize the solving effort for each run, we use linear elements (P_1 for simplex and Q_1 for rectangular elements). The meshes in comparison are: **Struc. Mesh** (the adaptive Cartesian mesh implementation presented in this chapter), **Unstr. Mesh** (the existing simplex mesh implementation), and **Peano Mesh** (the integrated regular Peano mesh). We compare for each simulation the memory storage of the actual mesh, and also the overall storage requirement. Since we choose such a simple PDE, the mesh storage in some cases becomes the dominant factor in the overall memory requirement. For more complex PDEs, with several unknown fields and higher order basis, the mesh storage becomes insignificant in comparison to the overall memory demand. Further, we also compare the total computation, mesh creation, and solving times. The solving time is the only time

[8] In this case, all the created Peano cells will be used.

that is not impacted by the mesh performance directly[9], but in all the other times, the mesh performance plays an important role. The tests were made on a desktop machine with 8GB RAM and with a 2.93GHz I7 Intel processor, and all runs were made in serial mode.

mesh	mesh storage	total storage	setup time	solver time	total time
Struc. Mesh	37	54	0.14	1.12	2.15
Unstr. Mesh	82	98	0.33	1.19	2.78
Peano Mesh	10	28	0.40	1.74	4.66

Table 6.1.: Storage required for the Poisson scenario measured in MB, and the runtimes measured in seconds. In the scenario, the Poisson equation is solved with linear elements in 2D. The spatial resolution is given by approx. 59000 quad elements (243×243)(twice as many triangles)

mesh	mesh storage	total storage	setup time	solver time	total time
Struc. Mesh	360	633	2.05	32.86	42.92
Unstr. Mesh	698	852	3.63	35.18	49.68
Peano Mesh	43	344	3.13	48.54	74.79

Table 6.2.: Storage required for the Poisson scenario measured in MB, and the runtimes measured in seconds. In the scenario, the Poisson equation is solved with linear elements in 2D. The spatial resolution is given by approx. 531000 quad elements (729×729)(twice as many triangles).

The results for the 243×243 resolution in 2D are presented in Tab. 6.1, and for a higher resolution of 729×729 the results are shown in Tab. 6.2. In 2D, the mesh storage requirement is the highest with simplex mesh, whereas the lowest is, as expected, with the Peano mesh. It is important to note that our Cartesian mesh implementation needs less than half of memory than the unstructured simplex mesh requires, even though it is a fully linearized tree, and allows random access. On the other side we can affirm that for both test cases, our Cartesian mesh implementation has the highest performance, by having the lowest overall runtime, outperforming even the simplex mesh.

Due to the trisection refinement in 3D, we were able to perform only one test, the next refinement level would have required parallel computing. The results for the $81 \times 81 \times 81$ resolution are shown in Tab. 6.3. The memory demand further increases for the simplex mesh (here with tetrahedron cells) and becomes the dominant factor, whereas the memory requirement for our linearized adaptive Cartesian mesh is less than 50% from the overall memory demand. In terms of runtime in 3D, our implementation outperforms again the other two variants, and the difference is even more significant than in 2D.

[9]Only indirectly by the matrix's sparsity pattern.

At the end of the comparison, we can summarize that the linearization of the Cartesian mesh within the frame of the Sundance toolbox pays off, both in terms of memory requirement, where the storage requirement does not become a dominant factor in the overall memory demand, and also in terms of mesh performance, where we outperform all the existing mesh types within Sundance.

mesh	mesh storage	total storage	setup time	solver time	total time
Struc. Mesh	783	1600	4.74	19.32	34.5
Unstr. Mesh	2700	3200	14.34	12.35	54.87
Peano Mesh	131	935	30.95	23.95	106.26

Table 6.3.: Storage required for the Poisson scenario measured in MB, and the runtimes measured in seconds. In the scenario, the Poisson equation is solved with linear elements in 3D. The spatial resolution is given by approx. 551000 brick elements ($81 \times 81 \times 81$)(four times as many tetrahedrons).

7. Fluid Flow with Nitsche's Method

This chapter introduces the required methods for the fluid flow simulation with Nitsche's method. It also shows computational benchmark results, which verify Nitsche's method in an IB context for the Navier-Stokes equations, even for near boundary values such as the measured drag and lift forces. In the previous chapter, we presented the first step towards IB method capabilities, the implementation of a parallel adaptive Cartesian mesh in the PDE toolbox Sundance. The next step towards a IB method capable PDE toolbox is the integration of special quadrature methods: **cut-cell integral** and **boundary integral** methods. In Chapter 4, we presented a variety of existing IB methods, and most of these methods, especially those who enforce the BC in a weak sense, have in common the necessity of a volume integral over Ω and a boundary integral over $\partial\Omega$. These features are also needed for Nitsche's method for the Navier-Stokes equations. Nitsche's method was already presented in detail in Chapter 4. Having an underlying Cartesian mesh that is non-conforming with respect to the boundary $\partial\Omega$, the first missing capability is to represent the boundary geometry independently of the underlying mesh. IB methods require an explicit boundary representation, since the boundary in this case is not represented by the facets of the Cartesian mesh. Therefore, we introduce first our geometry representation that we implemented and integrated in Sundance, with a focus on the developed modular interface that facilitates general geometry implementation in 2D and 3D. Then, we continue with the presentation of our newly developed cut-cell and boundary integration methods. The last section of this chapter presents a concrete application for the developed methods. We compute various fluid flow benchmark scenarios modeled by the Navier-Stokes PDE, where we impose the BC with Nitsche's method. We further compute the benchmark lift and drag coefficients for 2D and 3D scenarios in order to verify our approach and implementation. Parts of the methods and results presented in this section were already presented for 2D in [19]. In the following, however, we describe them in more detail and also extend them to 3D.

7.1. Boundary Geometry Representation

In this chapter, the main application is the Navier-Stokes PDE simulation with one particular IB method, Nitsche's method. Even though, at the end of this chapter, we apply the developed methods to one particular problem our goal is to introduce a general geometry representation that can be applied also to other IB methods within

Sundance. IB methods require an explicit boundary representation, and the geometry is non-conforming with respect to the mesh structure. This is illustrated in Fig. 7.1, where the computational domain Ω is embedded into a rectangular domain Ω_O. This way, the geometry can intersect cells in arbitrary ways, dividing the intersected cells into two parts. One part of these cells belongs to Ω, where the PDE shall be solved. The other part belongs to the fictitious domain Ω_F and should not be considered for the computations. To consider only the Ω part of an intersected cell for the integration of the weak form of a given PDE is one of the key features required for such IB methods.

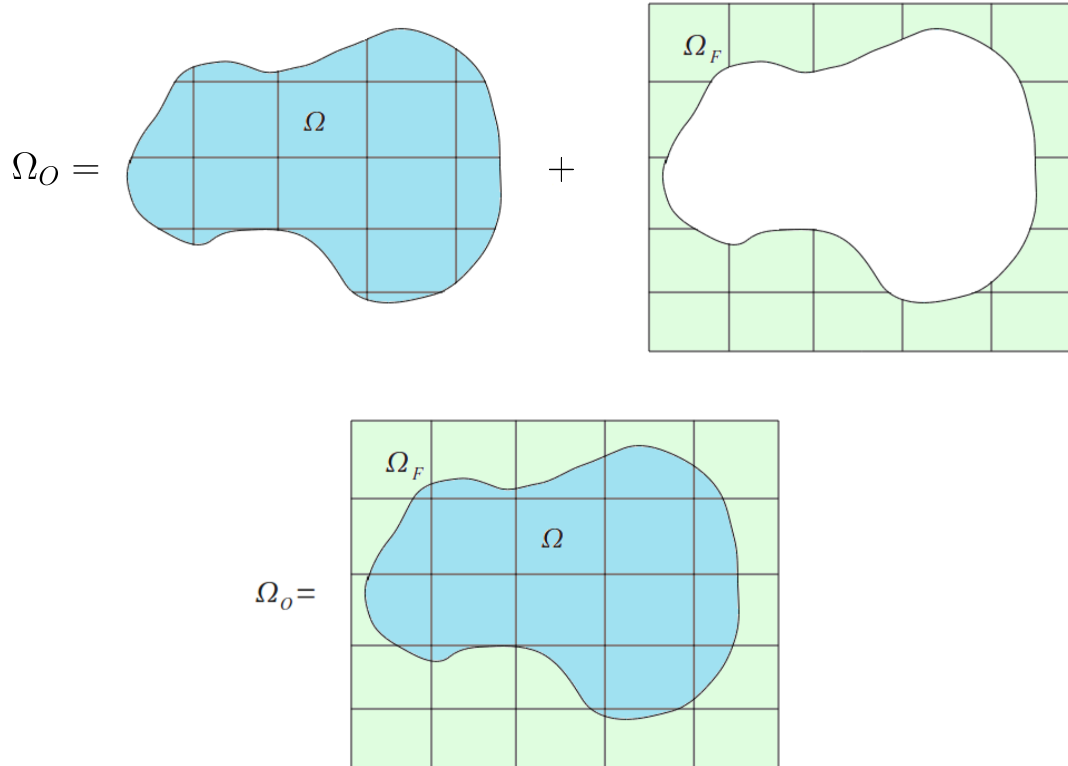

Figure 7.1.: Illustration of the IB method configuration including the underlying Cartesian mesh. The goal is to compute a PDE only on the computational domain Ω. This complex domain Ω is embedded into a rectangular domain Ω_O. This approach implicitly creates the fictitious domain Ω_F that should be neglected during computations.

7.1.1. Geometry Interface and Analytical Geometry Representations

Besides the **cut-cell integral**, Nitsche's method (and other IB methods) also requires **boundary integral** computations. This capability should be integrated in Sundance in a general way. These required features are illustrated in Fig. 7.2 in a cell-wise view, where the domain Ω and the respective boundary integrals need to be computed on such

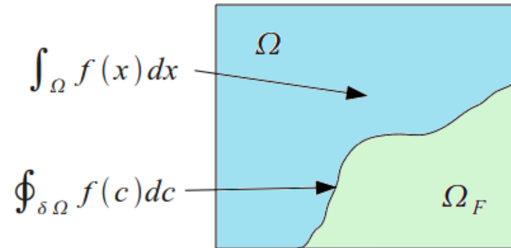

Figure 7.2.: Illustration of two required features from the cell-based view. The computation of integral $\int_{\Omega} f(x)\,dx$ and $\oint_{\partial\Omega} f(c)\,dc$ for a given function $f(x)$ and for the illustrated cell should be facilitated within Sundance. These are the two required features for Nitsche's method and also for other IB methods.

an intersected cells.

Both of the enlisted capabilities need an efficient geometry representation, since this geometry incorporates the definition of Ω and Ω_F. In the general case, it is difficult to integrate exactly on Ω and $\partial\Omega$. Therefore, the geometry should include, among others, functionalities that allow for the fast and efficient approximation of the boundary and domain within a given cell. The geometry should also provide point-wise information specifying if a given point is in Ω or in Ω_F. For these reason, we defined the general geometry interface for 2D and 3D by the following two functions:

1.) *geometry evaluation*: For a specified point in space, it returns the information whether this point is in Ω or in Ω_F. In the analytical description, this information is simply computed by the evaluation of the geometry equation. Hence, this functionality is called geometry evaluation.

2.) *line segment intersection*: In order to efficiently find an approximation of the Ω- and Ω_F-parts of an intersected cell, we require that the geometry returns all intersection point between a line segment and the geometry. The input line segments are usually the edges of the cells in 2D and 3D. The returned intersection points are the input for the cut-cell and boundary integral methods. A line segment in 2D and 3D might contain more than one intersection point. These cases have to be treated accordingly. This functionality could also be implemented indirectly with the first function (geometry evaluation) by using a bisection method. However, for higher efficiency, this functionality is delegated to the geometry, where it can be handled more efficiently.

These methods represent the general geometry interface that is consistent for 2D and 3D. However, there will be one exception for the later introduced polygon representation presented in the next section. In this case, the cut-cell and boundary integrals use special information that only a polygon in 2D can provide. The simplest geometrical representation is given by analytical expressions. By using such analytic formulas, one can

efficiently implement the presented two functions. This way, we implemented circle, rectangle, and ellipse objects in 2D, whereas we implemented sphere and brick objects in 3D. To illustrate, how the interface methods for such analytic geometries are implemented, we consider the circle in 2D. The analytic formula for this curve is $(x - o_x)^2 + (y - o_y)^2 = r^2$, whereas the equation of a line segment is $(x, y) = P_A + t\,(P_B - P_A)\,, t \in \mathbb{R}$, with the line segment defined by the points P_A and P_B. We further define for this example that the inside of the circle is Ω_F and outside the circle is the computational domain Ω. In this case, the evaluation of the circle implies computing the formula $(x - o_x)^2 + (y - o_y)^2 - r^2$. If the resulting value is positive, then the point (x, y) is in Ω, otherwise in Ω_F. For the intersection point, the line segment equation is inserted in the circle's equation resulting in a quadratic equation. The two real solutions, if there are any, are then tested, if they are on the line segment (between points P_A and P_B, e.g., $0 \leqslant t \leqslant 1$). The creation of such a geometry object is shown in Code 11, where the general geometry in 2D and 3D of Sundance is represented by the newly developed *ParametrizedCurve* object.

Code 11 Creating a circle and a box geometry object. For the circle object, the first two parameters represent the origin $(o_x, o_y) = (0.5, 0.5)$, the third parameter is the radius $r = 0.2$, and the last two parameters represent the weights (see Nitsche's method in Chapter 4) of the domains, in- and outside the circle. The second line shows the creation of a 2D box, where the first four parameters define the box and the last two parameters define the weight of the in- and outside domains.

```
ParametrizedCurve curveCircle = new Circle(0.5,0.5,0.2,1,1e-8);
ParametrizedCurve curve = new Box2D(0.3,0.3,0.4,0.4,1,1e-8);
```

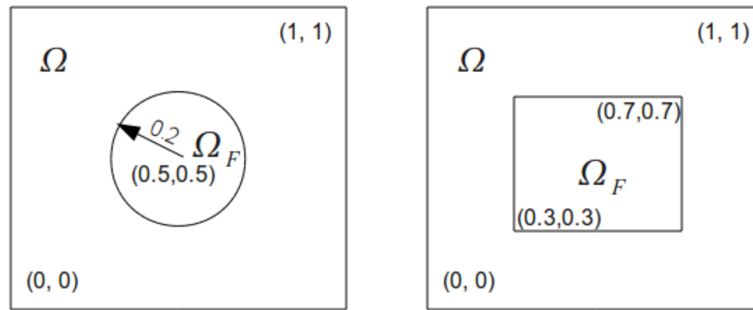

Figure 7.3.: Illustration of the created analytical geometries in Code 11.

In the example of Code 11, it is also illustrated that each domain in- and outside the geometry gets an assigned weight.[1] These weights determine which domain is the real computational domain (the one with coefficient $\alpha_1 = 1.0$) or becomes fictitious (weighted with $\alpha_2 = 10^{-8}$). In the results section, we show, that for numerical reasons Ω_F should not be weighted with zero, but only with a low α_2 number. These weights are stored accordingly in the geometry object, and are later used in the cut-cell methods.

[1] last two parameters, see Nitsche's method in Chapter 4

In the example of Code 11, we considered only closed curves in 2D. In general, the curves in 2D and surfaces in 3D do not have to be closed with respect to the computational mesh as long the two separate domains can be determined uniquely. Therefore, a plain or a line that are non-closed geometries could be potentially used as geometry objects in the computations.

Geometry based Cell Filtering

Based on the geometry information, the cells of the mesh are grouped into three cell filters. In Chapter 5, we presented the concept of *cell filters* that allows to treat a group of cells in a special way. In Fig. 7.4, we illustrate the usage of cell filters for an example geometry. The cells in Fig. 7.4 that are either completely inside (red) or outside (white) the geometry can be integrated in a classical way. The intersected cells that are marked with green in Fig. 7.4 have to be treated differently. Therefore, in the first step, the intersected cells have to be identified based on the geometry object. The identifications can either be based on the geometry evaluation or on the geometry intersection by a line segment. However, in 3D, not all the intersection cases can be determined with these two functions. We will come back to this problem in Section 7.2.2.

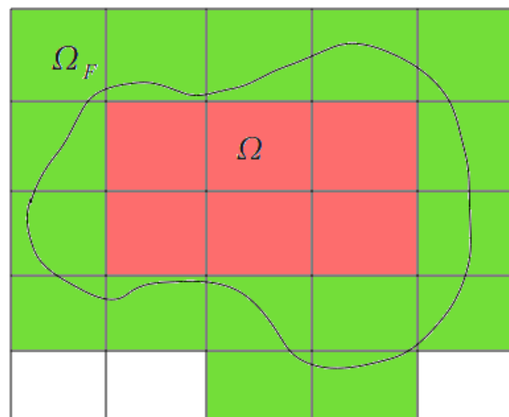

Figure 7.4.: A geometry divides the cells of the mesh into three groups. White color marks the cells that are completely outside the geometry, green cells are the cells that are intersected by the geometry, and the red cells are completely inside the geometry.

We highlight the Sundance code that creates the three cell filters in Code 12. The last line of Code 12 illustrates how such a cell filter is used for integration. The cell filter that delivers the intersected cells plays an important role, since only intersected cells are involved in the boundary and cut-cell integration within the IB context. This cell filter *OnCircle* in Code 12 combined with a special quadrature method allows to treat the intersected cells in a special way. Treating other cells (*InCircle* and *OutsideCircle* in

Code 12) in a classical way is important for efficiency, since those cells can be integrated without taking in consideration the boundary geometry.

Code 12 CellFilter declaration that is based on the geometry. The cells that are inside the geometry have negative evaluation values of the geometry, whereas cells that are on the other side have all positive geometry evaluation values.

```
ParametrizedCurve curve = new Circle(0.5,0.5,0.2,1,1e-8);
...
CellPredicate curveIN = new CellCurvePredicate( curve , Inside_Curve );
CellPredicate curveOUT = new CellCurvePredicate( curve , Outside_Curve );
CellPredicate curveON = new CellCurvePredicate( curve , On_Curve );
CellFilter InCircle = interior.subset(curveIN);
CellFilter OutsideCircle = interior.subset(curveOUT);
CellFilter OnCircle = interior.subset(curveON);
...
Expr int = Integral( OnCircle , ... , quad_spec , curve );
```

Even though a variety of boundary shapes can be described by basic geometrical objects such as circles and boxes in 2D and spheres and bars in 3D, there is a need for general and complex geometry representations in 2D and 3D. In the following, we present the polygon-based geometry representation in 2D and the triangular surface-based representation in 3D.

7.1.2. Polygons as Two-dimensional Geometry

A polygon is specified by a set of points and the line segments connecting them. A point in the polygon is allowed to be part of at most two line segments, and at least one. This way, a list of points defines a general polygon. The only information that remains to be specified is, if the polygon is closed or non-closed. A closed polygon implies, that the last point is connected to the first point forming a cyclic graph.

Both closed and non-closed polygons can represent a general boundary in 2D that is required for IB methods. Therefore, the next step is to present the implementation of the two geometry interface functions. For the *geometry evaluation*, it is required that the line segments have a consistent orientation. The orientation of a line is given by the start and end points of the line segment. In order to determine, whether a point is inside or outside a polygon, this orientation is crucial. A consistent orientation of the line segments is given, when the polygon is specified by a list of points.

We consider the illustration in Fig. 7.5, with three oriented line segments. The two evaluation points are the green points. For these two green points, it can be uniquely determined on which side of the polygon they are located. In order to compute this,

the first step is to locate the nearest line segment. Next, it needs to be determined on which side of the line the evaluation point is located. This can be simply done by a cross product of the line segment and the line segment, given by the start point and the evaluation point (marked with red in Fig. 7.5). The sign of the resulting cross product defines the sign of the evaluation of the polygon. Since the polygon can be

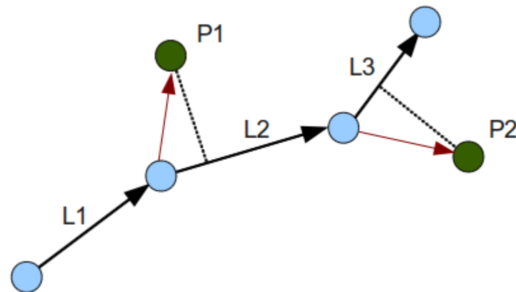

Figure 7.5.: Illustration of a simple polygon with three line segments and four points. The line segments are oriented in a consistent way, such that each point (e.g., the two green points) can be associated uniquely with one side of the domain.

composed of several hundreds of line segments, to determine the nearest line segment for an evaluation point becomes computationally costly in comparison with analytical geometry descriptions. A common way to reduce the computational overhead is to employ space-trees, which minimize the number of considered line segments [26]. In our implementation of the polygon in Sundance, we only employ a 'one cell' tree that consists of a singe cell. This cell is a bounding box, that contains the entire polygon. If a point evaluation is demanded outside the box, it can be computed in a simplified form as the distance to this bounding box. In many of our applications, the geometry covers only a small portion of the computational mesh. Therefore, significant computational effort is saved with this approach. To further increase the efficiency of the polygon's implementation, one should include a full space-tree implementation.

The second interface function represents the computation of the possible intersection points of a given line segment with the line segments of the polygon. The intersection point has to be located not just one the specified line segment, but also on the polygon's line segment, as this is illustrated by the green point I in Fig. 7.6. Similar to the previous function, the computation of possible intersection points turns out to be more computationally demanding then for analytically defined geometries. We also employ here the bounding box approach that limits the number of line segments for intersection point testing. If a specified line segment is completely outside of this bounding box and is also not intersecting the bounding box, then it has no intersection point with the polygon. The computation of the intersection points needs to be done in a robust way, such that all possible cases are treated. Therefore, we use a parametrized representation

of the line, where, according to Fig. 7.6, we write for the intersection point I

$$I = A + t\,(B - A)\,,\ t \in \mathbb{R},$$

and this intersection I point must also satisfy the cross product rule

$$(I - P1) \times (P2 - P1) = 0.$$

The resulting t should also satisfy the line segment (A, B) test such that $0 \leqslant t \leqslant 1$. Similarly, the resulting point I should be on the line segment $(P1, P2)$. The intersection point should be returned if it is contained in both line segments as illustrated in Fig. 7.6 by the green point I.

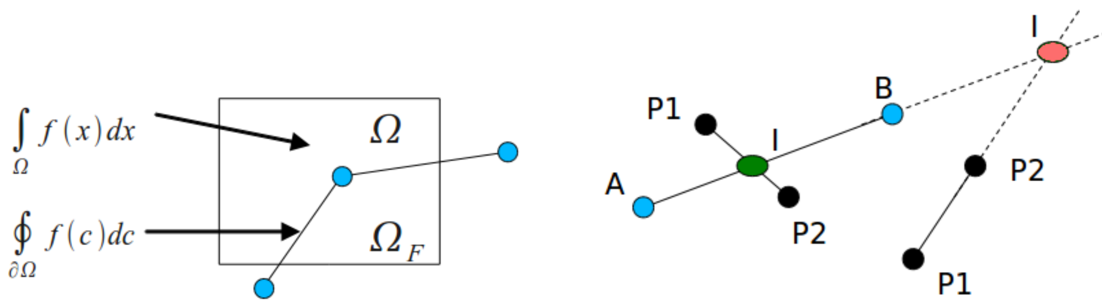

Figure 7.6.: Demonstration of the challenges of the boundary and cell integrals (left) that should take in consideration that the underlying geometry is a polygon. Illustration (right) of a polygon's line segment intersection (given by the points (A, B)) with the input specified line segment (defined by the points $(P1, P2)$). Only the intersection points that are contained in both line segments (green) should be returned.

On the left side of Fig. 7.6, we further illustrate, that the cut-cell and boundary integral methods could take into consideration that the underlying geometry is a polygon. In the case of analytical or general geometry representations, these integral methods would only approximate the real geometry. However, with polygons one could compute these integrals up to machine precision, since the geometry inside the cell consists of lines of segments.

One additional feature of the polygons, in contrast to analytical geometries, is the possibility to define nodal values that represent a function on this geometry. These values and functions can be used in Sundance expressions, but only as values on the right-hand side and not as unknowns. To include these nodal values as unknowns in the system, would require among others the extension of the matrix assembly method to multiple meshes with various dimensions. In the current implementation, there are various workarounds to consider these nodal values as unknowns in the systems.[2] Since each point of the polygon for a given function has one value assigned, we only allow linear basis functions

[2]partitioned methods, similar to the FSI formulations.

as illustrated in Fig. 7.7. The polygon in Fig. 7.7 is formed by four points $(I1, I2, I3, I4)$, and each of these nodes has an associated nodal value. In the case of the boundary integration, the values between the nodes are linearly interpolated. These nodal values of the polygon can be either set manually[3], or based on an *Integral* object[4] presented in Chapter 5.

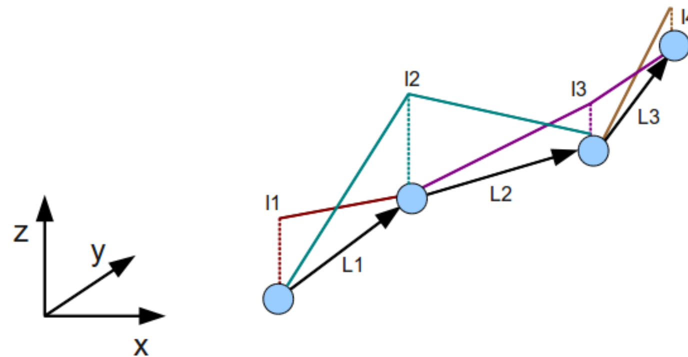

Figure 7.7.: Illustration of the polygon's nodal basis in 2D. The values between the polygon's points are interpolated linearly.

To illustrate the enlisted features of the developed polygon in Sundance, we consider Code 13. In the first line of this code, the polygon is created from an external file. This file consists of a list of points that form the polygon. The polygon is implicitly considered as closed, if this is not specified in the arguments. Similar to the previous geometry representations, the polygon contains the two weight factors of the in- and outside domain. In the second and third line, we add two scalar fields to the polygon, which together form a vector field. In the following four lines, we define the expression objects that later can be used in the boundary integral methods. For boundary integrals, we considered the example *uxD*vx + uyD*vy*, where *vx* and *vy* are test functions of a given problem. Next, we demonstrate the generality of the value setting feature of the polygon's implementation. The expression *ux_Eval* is used in this particular case to set the first scalar field of the polygon. In the final line of Code 13, the plotting of the polygon takes place. The scalar fields get plotted with the polygon structure. In 2D, scalar fields are pairwise coupled into a vector field, such that a vector field can also be potted within Sundance as illustrated in Fig. 7.8.

7.1.3. Triangle Surfaces as Three-dimensional Geometry

In 3D, as general surface representation we use triangular surfaces, where a continuous surface is composed of several triangles. Similar to 2D, a surface can either be closed

[3] based on coordinates or constant value

[4] a Sundance object that contains an expression and can be evaluated at each point of the mesh.

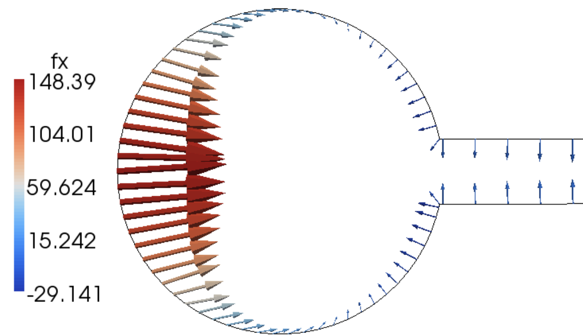

Figure 7.8.: Visualization of a polygon with an associated vector field.

Code 13 The usage of the polygon within Sundance. The first line illustrates the creation of the polygon. Then, linear functions are added to the polygon. Later, one can construct expressions that are used in integrations. The last three lines show the value setting of the polygon based on the *expression* object.

```
ParametrizedCurve polyg = new Polygon2D("polygon_file.txt",1.0,1e-8);
polyg.addNewScalarField( "velocityX" , 0.0 );
polyg.addNewScalarField( "velocityY" , 0.0 );
Expr x1 = new CoordExpr(0);
Expr x2 = new CoordExpr(1);
Expr uxD = new UserDefOp(List(x1,x2), rcp(new CurveExpr(polyg,0)));
Expr uyD = new UserDefOp(List(x1,x2), rcp(new CurveExpr(polyg,1)));
... Integral(OnCurve , ... uxD*vx + uyD*vy , ... );
Expr ux_Eval = Integral( OnCurve , expression , ... );
FunctionalEvaluator ux_Curve(mesh_Struct , ux_Eval);
polyg.setSpaceValues(ux_Curve , 0);
polyg.writeToVTK("polygon.vtk");
```

or non-closed. However, in our applications, we only employ closed surfaces. A closed surface is presented in Fig. 7.9, where a sphere is represented by 500 triangles. Such a surface is actually a 2D simplex mesh in a 3D context. In contrast to the mesh implementations in 2D, the triangular surface only needs to store the information regarding the points and triangles, since the edges do not provide additional information. Therefore, such a surface requires only the storage of positions of all points and the points corresponding to all triangles.

Analogously to 2D, the surface geometry implementation has to contain the two interface functions. The first function is the evaluation function that returns a value and represents the distance to the 3D surface. The sign of the returned distance defines, if the evaluated point E is inside or outside the geometry. In 2D and 3D, the definition of in- and outside can be flipped by calling the *flipDomains()* method of the geometry. To determine,

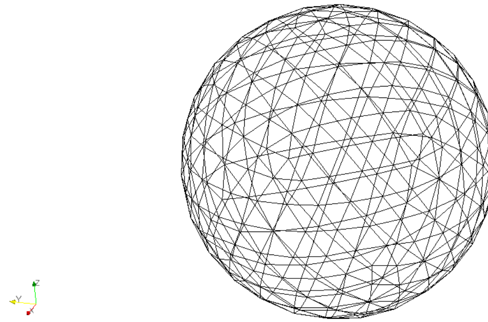

Figure 7.9.: An example of triangular surface representation, where 500 triangles approximate a sphere.

on which side the evaluation point E is located, we proceed in the similar way as in 2D for the polygons. The first step is to determine the nearest triangle. For a given triangle (A, B, C) in Fig. 7.10, this means taking the normal vector of the triangle.[5] Along this vector $V = (B - A) \times (C - A)$, we project E to the triangle's plain. If this projected point $E^{'}$ is outside the triangle, we consider the nearest point of the triangle to this projected point P, otherwise $P = E^{'}$. Then, the returned distance is just the distance between E and P. The crucial information, on which side of the triangle E is located, is given by the vector V. Point $E^{'}$ is expressed as $E^{'} = E + t \cdot V, t \in \mathbb{R}$. The sign of t decides, on which side of the triangle (A, B, C) E is located. A possible configuration is illustrated in Fig. 7.10, where the three points E, $E^{'}$, and P do not coincide. This method, to evaluate the surface for a given point E, was implemented in a robust way, such that all possible configurations of triangles (A, B, C) and E are handled in a consistent wav as demonstrated in Section 7.4.1 and in Chapter 8.

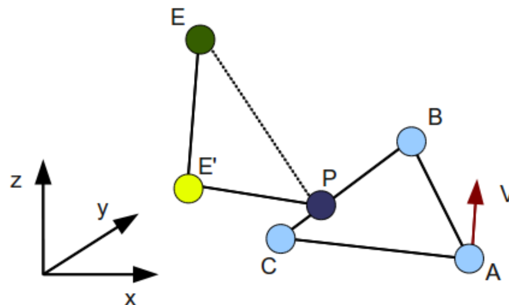

Figure 7.10.: Measuring the distance from a triangle. The normal vector V is crucial to determine on which side of triangle (A, B, C) the evaluation point E is located.

[5]The vectors form a triangle and the cross product of them gives a perpendicular vector to the triangle's plain.

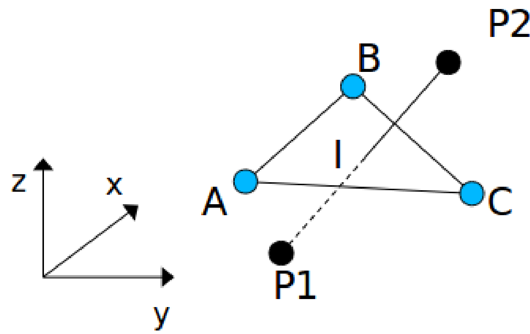

Figure 7.11.: Illustration of the intersection point of a triangle (A, B, C) surface and a line segment $(P1, P2)$.

At this point, we also mention, that it is important, that all triangles are oriented in the same direction. For the example in Fig. 7.9, this implies that all normal vectors point either inside or outside the sphere. If this is not respected, the geometry interface functions will return inconsistent results.

In 3D, the computational overhead to determine the distance to a triangle is more significant than in 2D for the polygons. Therefore, to reduce the number of triangles that need to be considered becomes even more important. The employment of space-trees in 3D would pay off even more significantly than in 2D. However, we employed only a one cell tree implementation that, similar to 2D, creates a bounding box. If the evaluation point E is outside this box, only the distance to this box is measured. Since in many of our applications in 3D, the surface covers only a small part of the computational mesh, and since we employ surfaces with less than a thousand triangles, this approach already reduces the computational effort dramatically. A further increase in performance of the geometry implementation could be expected if a full space-tree would be implemented.

The second function in the geometry interface is the computation of intersection points with a given line segment. This involves again the testing of several triangles for possible intersection points. Similar to the previous function, we also employ the bounding box. If both end points of a specified line segment are outside the bounding box and the line segment is not intersecting this bounding box, there are surely no intersection points. Otherwise, all triangles need to be tested for possible intersection points. To illustrate the underlying methods, we consider the example in Fig. 7.11. The input line segment is given by $(P1, P2)$ and the triangle is defined by the points (A, B, C). Intersection point I is computed by using the parametric description of the line $(P1, P2)$

$$I = P1 + t\,(P2 - P1)\,,\ t \in \mathbb{R}.$$

Point I must also satisfy the relation, which says that I lies within the plain of (A, B, C):

$$((B - A) \times (C - A)) \cdot (I - A) = 0.$$

In order to ensure, that I is on the line segment of $(P1, P2)$, it must hold $0 \leqslant t \leqslant 1$. Further, I must be inside the triangle (A, B, C) as shown in Fig. 7.11. Only then, I is a valid intersection point. This is simply tested by transforming the point I to barycentric coordinates of the triangle. Then the test, if I is inside (A, B, C), is evaluated in a straightforward manner.[6]

Similar to the polygon in 2D, triangular surfaces allow for the declarations and usage of multiple nodal values and functions. Defining values on a boundary interface could have multiple purposes. The most common purpose of this feature in our applications is to specify non-homogeneous Dirichlet boundary conditions on immersed boundaries in 3D.

Code 14 Usage of the integrated triangular surface in Sundance. In the first line, the surface is read from an external file. We further add scalar fields fx, fy, and fz that can be used in a boundary integration context. Finally, we set the first scalar field of the surface with a general expression *expr* and plot the surface.

```
//ParametrizedCurve triagSurf = new TriangleSurf3D("block.txt",1.0,1e-8);
ParametrizedCurve triagSurf
  = TriangleSurf3D::importGTSSurface("sphere5.gts",1.0,1e-8);
triagSurf.addNewScalarField( "fx" , 0.0 );
triagSurf.addNewScalarField( "fy" , 0.0 );
triagSurf.addNewScalarField( "fz" , 0.0 );
Expr fx = new UserDefOp(List(x1,x2,x3), rcp(new CurveExpr(triagSurf,0)) );
Expr fy = new UserDefOp(List(x1,x2,x3), rcp(new CurveExpr(triagSurf,1)) );
Expr fz = new UserDefOp(List(x1,x2,x3), rcp(new CurveExpr(triagSurf,2)) );
... Integral( OnCurve , -fx*vx - fy*vy - fz*vz , ... );
Expr ForceX = Integral( OnCurve , expr , ...);
FunctionalEvaluator ForceX_V( mesh , ForceX );
triagSurf.setSpaceValues( ForceX_V , 0 );
triagSurf.writeToVTK("Surface_3D.vtk");
```

We illustrate the integration of the presented triangulated surface representation into Sundance's descriptive language by Code 14. This code starts with the initialization of the surface that can be made from various formats. One can either use the internal format or a standard format such as GTS[7] and STL[8]. In the example of Code 14, the file with the internal format is stored in a TXT file. In the next lines of code, we define three scalar fields initialized with constant zero values. This is followed by the declaration of the expressions fx, fy, and fz, which represent the nodal value's expressions. Analogously to the 2D polygon case, the expressions fx, fy, and fz can be used in a boundary integral context, where in the corresponding lines, vx, vy, and vz represent the test functions of the underlying problem. The nodal values of the

[6] By checking the ranges of the resulting barycentric coordinates.
[7] http://gts.sourceforge.net/samples.html
[8] http://www.ennex.com/ fabbers/StL.asp

triangular surface can either be set directly based on coordinates[9] or based on a general evaluable expression *expr* in Code 14. In the final line of the example, the surface is plotted into a VTK file. During plotting, the scalar fields are bundled into vector fields. The resulting plot is presented in Fig. 7.12

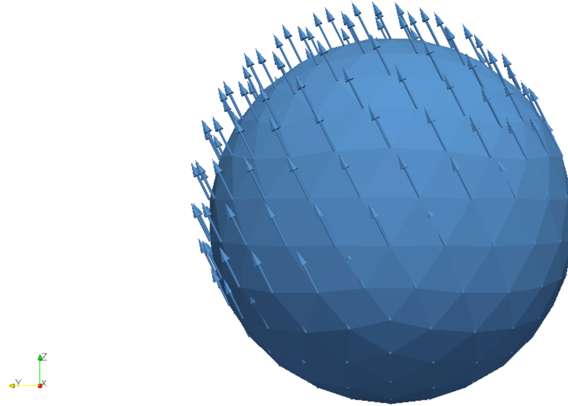

Figure 7.12.: Visualization of a closed triangular surface with a constant vector field.

7.2. Cut-Cell Quadrature

One of the necessities of IB methods, formulated in a weak form within Sundance is to have the capabilities of cut-cell integrations. This consists of treating the intersected cells in a special way by taking into account only the Ω part of each cell for the cell integration, whereas Ω_F is ignored. Until now, we presented various features of the boundary geometry implementations in 2D and 3D. In the following, we use these capabilities to develop a cut-cell method in Sundance for 2D and 3D. We also show the integration of the developed methods into the Sundance descriptive language. In all cases, the problem is restricted to the integration of an intersected cell. Therefore, in the following, we consider only one cell (E) in 2D and in 3D to present the methods.

The basic idea of the cut-cell method is to decompose such an intersected cell E into smaller subcells E_i, $i = 1, \ldots, M_E$, (basic cells such as triangles or quadrilaterals in 2D and tetrahedrons, prisms, or bricks in 3D) until the computational domain inside E denoted as $E \cap \Omega$ is represented exactly by these basic cells such that $E \cap \Omega = \cup_{i \in I_E} E_i$, $I_E \subset \{1, \ldots, M_E\}$. On these basic cells, the quadrature of the basis functions (test and unknown) can be done up to machine precision. The only detail to be defined is, how to decompose E into basic cells. This will be discussed specifically for a given dimension and geometry representation in Section 7.2.1 and Section 7.2.2.

[9]The geometry object offers access to the surface points and its nodal values as array.

Next, we look at the quadrature of the basic cells inside E. For given quadrature points $x_{i,j}$ and weights $\omega_{i,j}$ on the subcells $E_i \subset \Omega$ and a given function f (e.g., basis function), the quadrature is approximated as

$$\int_{E\cap\Omega} f(x)dx \approx \sum_{i\in I_E} \int_{E_i} f(x)dx \approx \sum_{i\in I_E} \sum_{j=1}^{N_i} \omega_{i,j} f(x_{i,j}). \tag{7.1}$$

Equation (7.1) provides a first quadrature rule on $E \cap \Omega$. In 3D, the number of basis functions f and the quadrature points $\sum_{i\in I_E} N_i$ can become large. Therefore, we reduce the number of quadrature points $\sum_{i\in I_E} N_i$, such that Sundance can use a smaller constant number of quadrature points for each intersected cell E. To reduce the number of quadrature points has also a technical aspect: For one cell filter Sundance allows only a constant number of quadrature points. This restriction is necessary to vectorize the assembly process of the system matrix in Sundance and for the efficient usage of the BLAS2 and BLAS3 routines.

To achieve this, we consider a set of geometry-independent quadrature points $p_k \in E$, $1 \leq k \leq K$, which are the quadrature points for cell E defined by a chosen quadrature rule. With the rule, the integration of the Lagrange polynomials l_k on $E \cap \Omega$ results in a quadrature rule $\int_{E\cap\Omega} f(x)dx \approx \sum_{k=1}^{K} w_k f(p_k)$, where the new weights can be precomputed as

$$w_k = \sum_{i\in I_E} \sum_{j=1}^{N_i} \omega_{i,j} l_k(x_{i,j}). \tag{7.2}$$

In (7.2), it is important that the quadrature rule on the E_i integrates all $l_k, k = 1, \ldots, K$, exactly. Here, we note that the resulting weights w_k, $k = 1, \ldots, K$, are computed only based on the geometry. We choose the Gauss-Lobatto [1] quadrature points $p_k \in E$, since this rule for a given order requires the same number of quadrature points and this allows us to span the Lagrange polynomials l_k, $k = 1, \ldots, K$ with the same order on E. In the computation of the special quadrature weights w_k, for a given basic cell E_i, we use Gauss-Legendre [1] quadrature (for $\omega_{i,j}$) that, especially in 3D, uses considerably less quadrature points than the corresponding Gauss-Lobatto rule.

This way, for each intersected cell E, we precompute the set of special weights $w_k, k = 1, .., K$ that is only used for E. Since one set of weights is specific to one cell and only depends on the boundary geometry, these weights are stored in the mesh object.[10] However, if the geometry is changed, all these weights need to be recomputed for all intersected cells. The advantage of the proposed method is the separation of geometry and function, such that quadrature weight precomputation is possible based only on the geometry.

According to the method proposed above, the function f is integrated only over $E\cap\Omega$. If $\int_{E\cap\Omega} 1dx \ll \int_E 1dx$, the method potentially could induce a numerical singularity in the

[10] The extension of the mesh interface for this purpose is marginal, therefore, it is not presented here.

system matrix since the area of $E \cap \Omega$ is numerically zero. To avoid such singularities, we use the weighting factors that we already introduced for the geometries (see also Nitsche's method in Chapter 4). These factors weigh the domains Ω and Ω_F and we denote them as α_1 weighting Ω and α_2 weighting Ω_F. They are chosen by the user, since they can be specific to applications. The integral with the weighting factors $\alpha_1, \alpha_2 \geq 0$ is written as

$$\alpha_1 \int_{E\cap\Omega} f(x)\,dx + \alpha_2 \int_{E\setminus\Omega} f(x)\,dx = (\alpha_1 - \alpha_2) \int_{E\cap\Omega} f(x)\,dx + \alpha_2 \int_E f(x)\,dx. \tag{7.3}$$

From (7.3) results the computation of the modified weights w_k^m associated to the Lagrange polynomial l_k

$$w_k^m = (\alpha_1 - \alpha_2) w_k + \alpha_2 \sum_{j=1}^{N} \omega_j l_k(x_j) = (\alpha_1 - \alpha_2) w_k + \alpha_2 \int_E l_k(x)\,dx, \tag{7.4}$$

where on the cell E, the polynomial l_k is integrated up to machine precision with N quadrature points. By using formula (7.4) on cell E, Ω has a weight of α_1, and Ω_F is weighted by α_2. In our applications, we set the weight factors usually to $\alpha_1 = 1.0$ and $\alpha_2 = 10^{-8}$.

Finally, we add a practical aspect to the developed method. In some cases, it is more feasible to approximate the Ω_F part of E. We denote the set of basic cells with $E \cap \Omega_F = \cup_{i \in I'_E} E'_i$, $I'_E \subset \{1, \ldots, M'_E\}$. Then, the weight w_k is computed as

$$w_k = \int_E l_k(x)\,dx - \sum_{i \in I'_E} \sum_{j=1}^{N_i} \omega_{i,j} l_k(x_{i,j}). \tag{7.5}$$

In the following, we present the boundary discretization in 2D and 3D for the presented curve and surface representations inside the cell E that consist of a decomposition of $E \cap \Omega$ or $E \cap \Omega_F$ into basic integrable cells. These basic cells are used in (7.2), (7.4), and (7.5) to compute the modified weights $w_k, k = 1, \ldots, K$, which once computed can be reused and stored in the mesh.

7.2.1. 2D Cut-Cell Integration Method

In the first step, we only consider the 2D general or analytical representation, where we restrict ourselves to cases, where one cell is cut no more than twice and one edge is cut no more than twice. These conditions define the **regular case** in 2D. These restrictions are valid for all of the regular cases, or this case can be achieved by either additional refinement of the mesh or simplifications. The general geometry representation includes also the polygon in 2D and analytical curves.

We start with the cell view of the problem illustrated for three cases in Fig. 7.13. The represented cases are only meant to illustrate the main idea behind the cut-cell integration with general boundary representations. The intersection points on the edges of the cell are given by the general geometry representation, and the geometry is approximated by a line inside cell E. Inside a cell, the geometry is approximated by a line. Therefore, we need at most one triangle and one quad cell to approximate the $E \cap \Omega$ part of E. On the left side of Fig. 7.13, even a single triangle T_1 approximates Ω_F. The second case in Fig. 7.13 needs the triangle T_1 and the quad cell Q_1 to approximate Ω in E. In Fig. 7.13 we show one irregular case that can occur even with relatively refined meshes and with sharp boundary corners. Therefore, we treat this case accordingly, by ignoring the edges that are intersected twice by the geometry. In such a way, the resulting approximation consists of a line, and $E \cap \Omega$ is approximated by the triangle cell T_1 and the quad cell Q_1. Treating such irregular cases ensures the robust implementation of the cut-cell method for complex geometries.

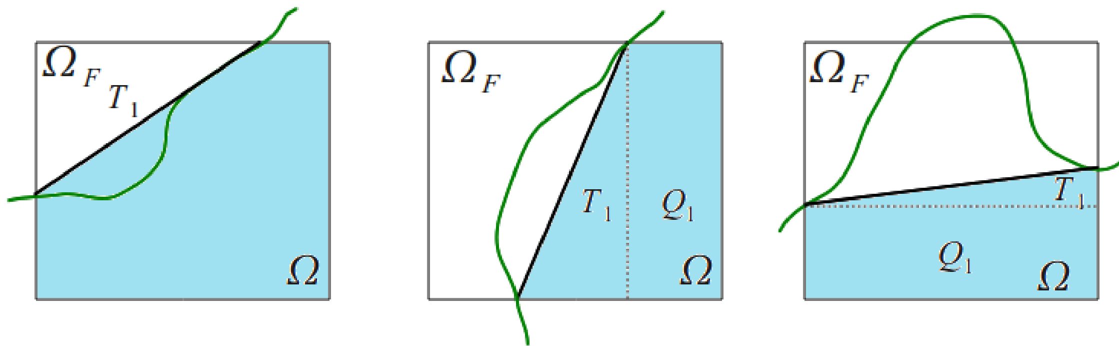

Figure 7.13.: Illustration of a cut-cell in a general geometry. The boundary of a general or analytical geometry, marked with green color, is interpolated by a line (black color) inside the intersected cell E. The first two cases (left middle) represent regular cases, where there are only two intersection points in total. The third case (right) is an irregular one with four intersection points that is also approximated by a single line. Two intersection points on one edge are simply ignored, transforming it into a regular case.

In general terms, the cut-cell method for general geometry representation simplifies the irregular cases by ignoring the edges that are intersected twice. The resulting configuration is a regular one that contains only two intersection points. Such a regular case can always be decomposed in one triangle and one quad cell[11] that are used for the special weight computations.

[11] Or decomposed in one triangular cell.

2D Cut-Cell Integration Method for Polygons

The developed 2D cut-cell method for general and analytical geometry representation also fits for polygons. However, for polygons, we developed a special cut-cell method that takes into consideration the underlying geometry representation. Assuming, that a polygon represents a realistic scenario by fulfilling the cone condition [21], the polygon cuts a cell at most four times. Cases when a cell is intersected more than four times are ignored here. For these cases, additional mesh refinements are required.

Knowing, that the underlying geometry is a polygon, we can acquire the polygon's points that are inside a given cell E. The polygon has been extended with this functionality that does not cost additional computational overhead.[12] With this information, the cut-cell integration can be computed with up to machine precision as illustrated by the three cases in Fig. 7.14. By using the polygon's points inside cell E, one can decompose $E \cap \Omega$ or $E \cap \Omega_F$ into a set of triangles, quads, and trapezoid cells. In the first case of Fig. 7.14, $E \cap \Omega$ is represented by the triangle T_1 and by the trapezoidal cells Tr_1 and Tr_2. The second case approximates $E \cap \Omega_F$ with two trapezoidal cells, and by using formula (7.5), one can also approximate $E \cap \Omega$. The third example in Fig. 7.14 represents a case with

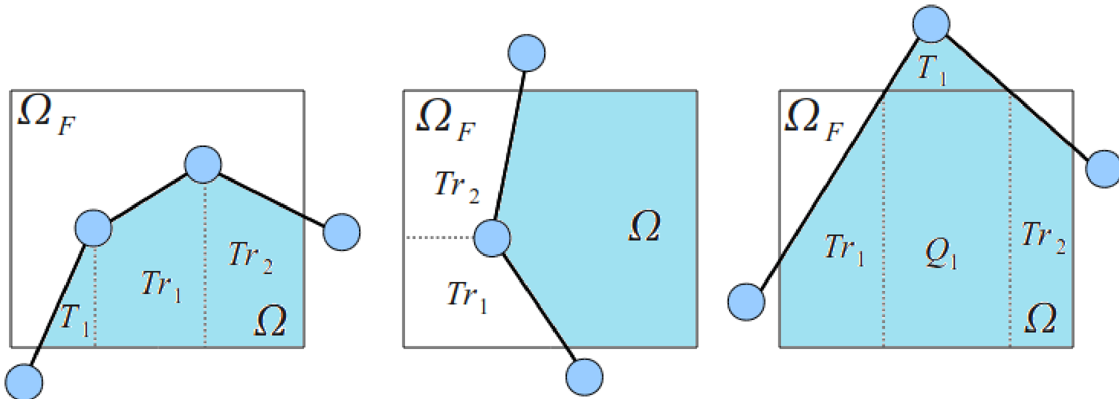

Figure 7.14.: Illustration of cut-cell method for polygon. The first two cases (left and middle) represent cases with two intersection points, where $E \cap \Omega$ or $E \cap \Omega_F$ is approximated by a set of triangle and trapezoid cells. The third case (right) shows a case with four intersection points that is also treated accordingly.

four intersection points.[13] By using the intersection points and the polygon's internal points, these cases can also be handled accordingly, by decomposing $E \cap \Omega$ into quad cell Q_1 and trapezoidal cells Tr_1 and Tr_2. The triangle cell T_1 suggests that the cell above can also be integrated up to numerical precision, as the geometry is not approximated

[12] In the polygon setup phase, each point gets an assigned cell LID, and the points that are not on the mesh domain will have -1 associated LID.
[13] was an irregular case previously.

by only one line.

In general, the set of basic cells to represent $E \cap \Omega$ or $E \cap \Omega_F$ can be found by projecting the polygon's points to one of cell's edges in x- or y-direction. This edge has to be chosen such that the projected polygon points represent a monotone increase in the edge's direction. This monotony condition for the x-direction is fulfilled in the left and right case in Fig. 7.14, whereas the y-direction fulfills this condition in the middle case of Fig. 7.14. Therefore, the projections are made according to this criterion. Once such an edge is found, the projected points with the polygon's and intersection points form the basic cells that represent the decomposition. If none of the edges satisfies this monotony condition, as a default an edge in the x-direction is chosen as projection edge. This approach gives wrong integration results only with multiple sharp edges within a cell that can be resolved by additional mesh refinement.

The robust implementation of the cut-cell method for polygons will be demonstrated for various applications in Section 7.4.1 and Chapter 8.

7.2.2. 3D Cut-Cell Integration Method

Similar to the polygon specific cut-cell method in 2D, one can implement a triangular surface based cut-cell method. This would imply the implicit usage of the triangles' points and the treatment of all possible intersection cases. In 3D, the intersection configuration can occur, where the geometry intersects only a surface and none of the edges of the cell. In addition, all the possible intersections of the edges should be treated similar to the Marching Cubes Method [63]. Due to the complexity of such a triangulation specific integration, we treat in 3D all geometry representation in the same way and do not use geometry specific information as we do in 2D for polygons. The main goal is here to have a robust approximation method of the cut-cell integration in 3D, such that all possible cases are treated and no breakdown of the simulation happens.

We only consider cases where a given cell's edges are intersected by the geometry. The intersection points are the main information to approximate the $E \cap \Omega$ part of the cell. Similar to the 2D case, if an edge is intersected twice, these two intersection points are ignored. If a face (quad) of a cell is intersected more than twice, further simplifications are necessary, such that an approximation of the intersection surface can be determined. An intersected cell that fulfills the conditions above we call **regular case** in 3D. Once having all the intersection points of the edges, the intersection surface needs to be determined. This intersection surface is illustrated in Fig. 7.4 for one possible configuration of intersection points.

We approximate the intersection surface with simplex cells. Therefore, a triangulation of this surface is necessary. The triangulation inside a brick cell has its restrictions, since the intersection points need to be connected along the side faces of the brick cell, such

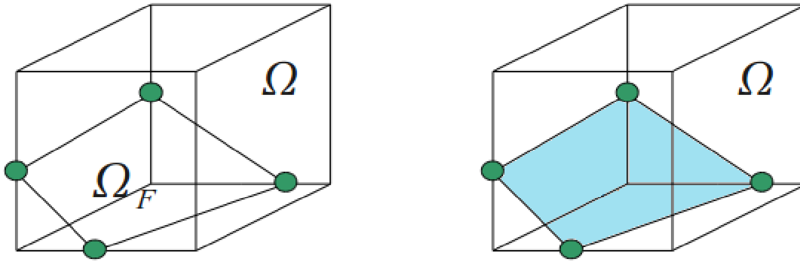

Figure 7.15.: Illustration of one possible intersection point configuration in 3D (left). The intersection surface is approximated by the blue surface (right). Such an intersection surface needs to be triangulated in the first step.

that the intersection surface is represented accurately. Since we consider only the regular cases, a cell can have at most six intersection points and at least three. Therefore, the number of resulting triangles varies from one to four. Possible intersection configurations and the resulting triangular surfaces are presented in Fig. 7.16. The first step towards the triangulation is to form a closed polygon with the intersection points, where the lines have to be contained in the cell's edges. Once this polygon is formed, the next step is to form a triangulation out of this polygon that can be done in a straightforward manner as shown in Fig. 7.16. The resulting triangulation is further used in the boundary integral that is discussed in Section 7.3.

For the cut-cell integration, we need to decompose $E \cap \Omega$ into basic 3D cells. This method is more complex than in 2D with polygons, where the intersection and polygon's points are projected to one of the cell's faces. In 3D, the triangulation with the intersection points needs to be projected to one of the cell's faces in the directions (x, y), (x, z), and (y, z). The criterion for the chosen face is that the projection should result in a non-overlapping set of cells. Since we are not considering any internal points, there is always a projection face that satisfies this criterion. This face can be chosen based on the edges that are intersected. If we consider the top left example in Fig. 7.16, we notice, that this surface can be projected to any of the faces, since edges which are orthogonal to the given face have been intersected. Considering the top-right case in Fig. 7.16, we can state similarly, that the resulting triangulated surface can be projected in both (y, z) and (x, z), but it should not be projected on one of the (x, y) faces, because it does not intersect any edge that is orthogonal to these faces. For the bottom-left and bottom-right example, we can apply the same criterion. These surfaces can be projected to any of the faces, since they intersect edges that are orthogonal to these faces. This observed rule is applied to all intersected cells in order to determine the projection face, where the intersection points and the intersection surface is projected to.

The next step is to build a decomposition of the cell to approximate $E \cap \Omega$ or $E \cap \Omega_F$, based on the projected intersection surface. Previously, we showed how the intersection

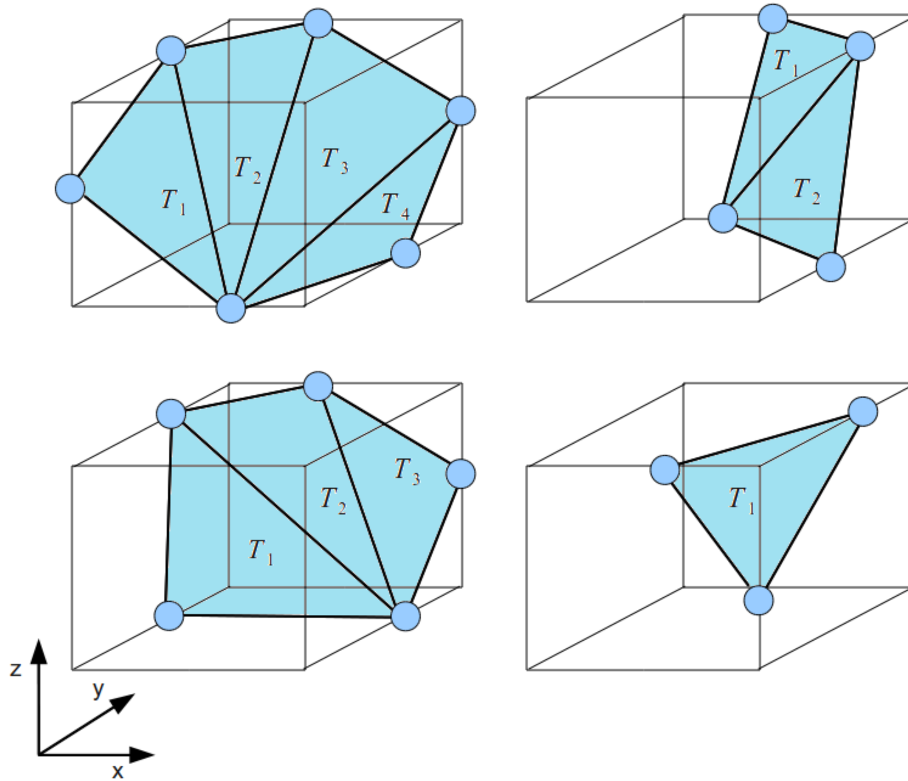

Figure 7.16.: Illustration of possible intersection configuration of the brick cell. The resulting triangulation is also shown, where the triangles are denoted by $T_1, \ldots, T_n$. It is important that the triangulated surface represents the intersection surface consistently. The number of triangles n varies from one to four.

points and the resulting triangulation of the intersection surface of the cell get mapped to one chosen face. Using the mapped points, the goal is to build basic cells in this plain that cover the face. This process is illustrated on the left side of Fig. 7.17, where the blue circles represent the intersection points from the top-right example of Fig. 7.16. The intersection points are projected to the (x, y) face, and are marked by green colored circles. In addition to the projected points, one corner point is required to complete the $E \cap \Omega$ part of the intersected cell. This additional point and its projection are marked also with green circles. In such a way, the intersection points with their projections and with the additional points form three basic cells: one tetrahedron and two prism cells, which can be integrated up to machine precision.

This idea of constructing basic 3D cells from the plane projection and additional points can be generalized. Once these basic 3D cells are constructed, the last step in the 3D cut-cell method is the integration of these cells according to the presented equations (7.2), (7.4) and (7.5).

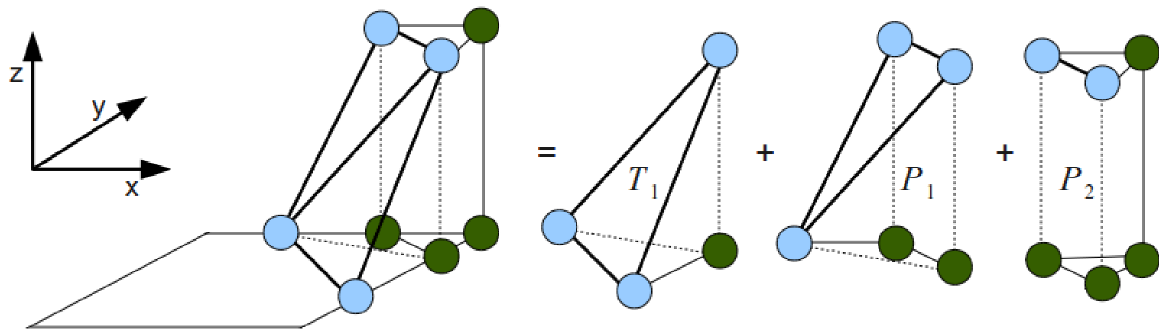

Figure 7.17.: Illustration of the $E \cap \Omega$ decomposition into basic cells. This figure illustrates the decomposition of the example in Fig. 7.16 (top-right). The triangular surface is projected to the (x, y) face resulting in three basic cells: one tetrahedron T_1 and two prism cells P_1 and P_2. The green points represent the projected points and the points that are needed to make these cells complete.

7.2.3. Cut-Cell Integration Methods in Sundance

In this section, we illustrate the integration of the developed cut-cell methods in 2D and 3D into Sundance's descriptive language. We do not focus on the software design challenges that we faced by the integration of these cut-cell methods into the simulation pipeline of Sundance (presented in Chapter 5), but only on the developed user interfaces of the implemented methods in Sundance.

In Code 15, we illustrate the usage of the presented cut-cell methods for 2D. The first geometry created is an analytical representation, whereas the second one is a polygon created from an external file. First, we initialize a cell filter that selects only the intersected cells, such that the non-intersected cells can be integrated in a more efficient way. Then, we create the Gauss-Lobatto quadrature rule that allows for the usage of the developed methods. In the last two lines of Code 15, we define the cut-cell integrals by specifying this particular quadrature and the geometry. It is crucial, that the geometry appears as last argument in the integral object, otherwise it means a 'regular' integral over the selected cells. The integral computed by this method is $\int_{E \cap \Omega} \nabla u \nabla v \, dx$, where u is an unknown function and v is a test function. In 3D, the usage of the developed cut-cell methods is done in a similar way as shown in Code 16. A triangular surface is initialized from an external file. With the Gauss-Lobatto quadrature rule, it is input argument for the *Integral* constructor, selecting the cut-cell method for the intersected cells in 3D.

Code 15 Illustration of the cut-cell method integration into Sundance's descriptive language for 2D. After the two types of geometries are created, we initialize the cell filters to select only the intersected cells. In the last two lines, we define the cut-cell integration by specifying the quadrature method and the geometry as last arguments.

```
ParametrizedCurve circle = new Circle(0.5,0.5,0.2,1.0,1e-7);
ParametrizedCurve polygon = new Polygon2D("polygon.txt",1.0,1e-7);
CellFilter Omega = new MaximalCellFilter();
CellPredicate circleON = new CellCurvePredicate( circle , On_Curve );
CellFilter OnCircle = Omega.subset(circleON);
CellPredicate polygonON = new CellCurvePredicate( polygon , On_Curve );
CellFilter OnPolygon = Omega.subset(polygonON);
QuadratureFamily quad_hi = new GaussLobattoQuadrature(6);
Expr eqnC = Integral(OnCircle,(grad*u)*(grad*v),quad_hi,circle);
Expr eqnP = Integral(OnPolygon,(grad*u)*(grad*v),quad_hi,polygon);
```

Code 16 Illustration of the cut-cell method integration into Sundance's descriptive language for 3D. The cut-cell method is used for a triangular surface by specifying the geometry and the quadrature method as last arguments in the *Integral*'s constructor.

```
ParametrizedCurve geometry = new TriangleSurf3D("cylinder.txt",1,1e-8);
CellFilter Omega = new MaximalCellFilter();
CellPredicate geometryON = new CellCurvePredicate(geometry,On_Curve);
CellFilter OnGeometry = Omega.subset(geometryON);
QuadratureFamily quad_hi = new GaussLobattoQuadrature(6);
Expr eqnG = Integral(OnGeometry,(grad*u)*(grad*v),quad_hi,geometry);
```

7.3. Curve and Surface Integrals

The next requirement for IB methods, which are formulated in a weak form, is the capability to compute boundary integrals. In the IB case, the boundary is inside a given intersected cell E. This approximation of the boundary must be the same as the one used for the cut-cell method! This assures the implementational consistency of a given IB method. For the Navier-Stokes equations, an inconsistent approach leads to distorted boundary solutions.

Thus, we use the same boundary discretization that we presented for the cut-cell method in 2D and 3D. The main idea is again, to decompose the boundary inside a given cell E in such a way, that the boundary inside E, denoted by Γ_E, is given by $\Gamma_E = \cap_{i \in K_E} B_i$, where $K_E \subset \{1, \ldots, L_E\}$ is the set of basic cells containing L_E elements. With this

approximation, the boundary integral can be written as

$$\oint f(c)\,dc \approx \sum_{i\in K_E} \oint_{B_i} f(c)\,dc \approx \sum_{i\in K_E} \sum_{j}^{N} \omega_{i,j} f(c_{i,j})\,. \tag{7.6}$$

Formula (7.6) requires $L_E \times N$ quadrature points, where, with N quadrature points, f is integrated on B_i exactly. Similar to the cut-cell methods there is the possibility to reduce the number of quadrature points by geometry-dependent precomputation of weights. However, for general boundary integrals, which can contain normal vector expressions, this turns out to be complicated. In addition, there might be specific expressions that need to be evaluated in the neighborhood of the boundary (e.g., nodal values of a triangular surface). Therefore, for each intersected cell, we store $L_E \times N$ quadrature points. We also point out, that, in contrast to the cut-cell method's decomposition, the boundary integrals always decompose the boundary into simplex cells, where the maximal number of simplex cells within E is small. Therefore, we can use the same number of quadrature points for the boundary integral on any intersected cell. We recall, that a constant number of quadrature point is required by the Sundance assembly process. In addition, normal vector components might be required for specific boundary integrals and need to be computed. The quadrature points for the boundary integrals and the corresponding normal vector components are stored in the mesh object. The only important detail that remains to be solved is to find the discretization of the boundary $\Gamma_E = \cap_{i\in K_E} B_i$. This is specific to dimensions and to geometry representation. In the following, we present discretizations of the boundary for 2D and 3D.

7.3.1. 2D Curve Integration

In 2D, we reuse the discretization that we already presented for the cut-cell method for the boundary integration. For analytical or general boundary representation, the curve is always approximated by a line within an intersected cell E as shown in Fig. 7.13. This discretization is illustrated on the left side of Fig. 7.18, where a given function f needs to be integrated only along the illustrated line segment. Therefore, for such boundary integrals, any quadrature rule might be used that allows for the integration on a line. The normal vector per definition is pointing outwards of Ω, and is constant for the quadrature points on this line.

However, if the geometry is represented by a polygon, we use the consistent representation from the cut-cell method. This boundary discretization is shown on the right side of Fig. 7.18, where the boundary is discretized with a set of line segments. Each line segment can be integrated exactly. The only question left open is, how many line segments can be inside a cell. The number of line segment varies among the intersected cells and depends on the resolution difference between the mesh and the polygon. Since the number of quadrature points has to be constant for each intersected cell, we limit the maximal number of line segments. This number can be set by the user and is denoted

by L_E. If a cell contains $l < L_E$ line segments, the quadrature weights $\omega_{i,j}$ of the last $L_E - l$ line segments are set to zero. In this way, only the first l line segments are taken into consideration. In addition, for each line segment, the corresponding normal vector is computed as illustrated in Fig. 7.18.

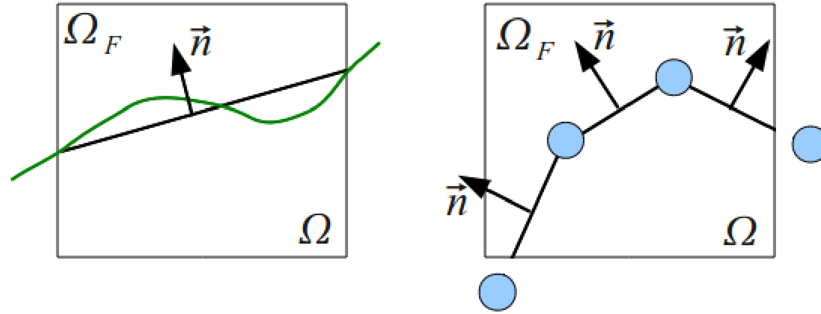

Figure 7.18.: Illustration of the boundary integration in 2D. For analytical or general geometry representations we discretize the boundary as a line segment (left), whereas with polygons (right), we integrate each line segment within the cell. The normal vectors **n** are pointing outwards of Ω.

7.3.2. 3D Surface Integration

For the 3D boundary integral, we recall the triangular surface discretization from the 3D cut-cell method. We use the same boundary discretization in 3D for the surface integrals, and the simplex cells are triangles. In the presented boundary discretization, the number of triangles varies from one to four for the regular cases. Similar to the polygon curve

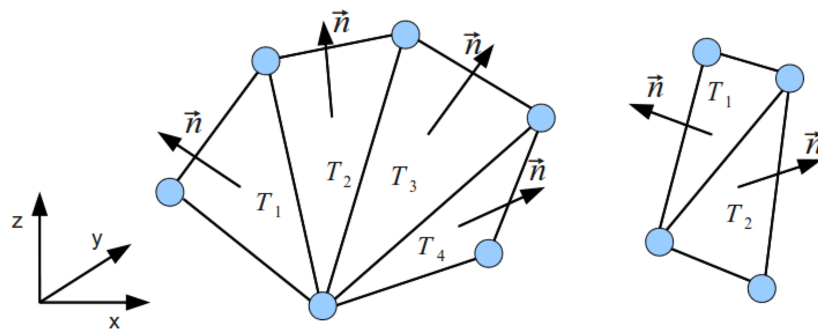

Figure 7.19.: Two examples of the boundary surface integration in 3D. The intersection points form a triangulated surface $T_1, \ldots, T_l, l < 5$ that represents the boundary surface. Each triangle has its own normal vector **n**.

quadrature, we use a constant number of quadrature points for intersected cells. $L_E = 4$

is set to the maximum number of triangles. If a cell has less than four triangles $l < 4$ as intersection surface, the quadrature weights $\omega_{i,j}$ of the last $4 - l$ triangles are set to zero. For 3D surface integrals, the normal vectors also need to be computed and stored for each quadrature point. Such a normal vector is constant for a given triangle as illustrated for two cases in Fig. 7.19.

7.3.3. Curve and Surface Integral Implementations in Sundance

The developed boundary integral methods have been integrated into Sundance. In this section, we focus solely on the integration of the developed methods into the problem description language of the toolbox. The software design challenges to integrate modularly these capabilities into the assembly process of the matrix are not presented here. In all cases, the key step to trigger boundary integrals is the usage of the *ParamCurveIntegral* class that wraps a given geometry object as illustrated in Code 17. In the following, we present code sections that use the presented boundary integrals to compute $\oint (\mathbf{n}\nabla u)\, v dc$, where u is the unknown function, v the test function, and $\mathbf{n}$ is the normal vector. The usage of the normal vector as an expression has also been accomplished. The user can access the components of the normal vector by the *CurveNormExpr*() object specifying the dimension index in the constructor.

Code 17 Sundance code of the 2D boundary integral with analytical geometry representation.

```
ParametrizedCurve circle = new Circle(0.5,0.5,0.2,1.0,1e-7);
ParametrizedCurve circleBInt = new ParamCurveIntegral(circle);
CellFilter Omega = new MaximalCellFilter();
CellPredicate circleON = new CellCurvePredicate( circle , On_Curve );
CellFilter OnCircle = Omega.subset(circleON);
QuadratureFamily quad_c = new GaussianQuadrature(4);
Expr nx = new CurveNormExpr(0);
Expr ny = new CurveNormExpr(1);
Expr eqnC = Integral(OnCircle,nx*(dx*u)*v+ny*(dy*u)*v,quad_c,circleBInt);
```

For general or analytical geometry representations in 2D, one can use any given quadrature class, since the boundary inside a cell is approximated by a single line. The corresponding user code is shown in Code 17. The structure of the user code is similar for all three boundary integral methods. In the first two lines, the geometry and the corresponding wrapper *ParamCurveIntegral* are created. Once the cell filter for the intersected cell is created, we further declare a quadrature rule for the boundary integral that, in this case, can be any accessible quadrature rule. Next, we create the expressions that represent the normal vector's components. In the last line of Code 17, we create the boundary integral object. It is crucial, that, as last argument, we specify a

ParamCurveIntegral wrapper object of the given geometry. This determines, that this is a boundary integral and not a cut-cell integral.

Code 18 Sundance code of the 2D boundary integral with a polygon. For this case, one needs to use the special quadrature method *PolygonQuadrature*.

```
ParametrizedCurve polygon = new Polygon2D("polygon.txt",1.0,1e-7);
ParametrizedCurve polygonInt = new ParamCurveIntegral(polygon);
CellFilter Omega = new MaximalCellFilter();
CellPredicate polygonON = new CellCurvePredicate( polygon , On_Curve );
CellFilter OnPolygon = Omega.subset(polygonON);
QuadratureFamily quad_g = new GaussianQuadrature(4);
QuadratureFamily quad_c = new PolygonQuadrature(quad_g);
PolygonQuadrature::setNrMaxLinePerCell(10);
Expr nx = new CurveNormExpr(0);
Expr ny = new CurveNormExpr(1);
Expr eqnC = Integral(OnPolygon,nx*(dx*u)*v+ny*(dy*u)*v,quad_c,polygonInt);
```

Code 19 Sundance code of the 3D surface integral, where one needs to use the special quadrature class *SurfQuadrature*.

```
ParametrizedCurve geometry = new TriangleSurf3D("cylinder.txt",1,1e-8);
ParametrizedCurve geometrySInt = new ParamCurveIntegral(geometry);
CellFilter Omega = new MaximalCellFilter();
CellPredicate geometryON = new CellCurvePredicate( geometry , On_Curve );
CellFilter OnGeometry = Omega.subset(geometryON);
QuadratureFamily quad_g = new GaussianQuadrature(4);
QuadratureFamily quad_s = new SurfQuadrature(quad_g);
Expr nx = new CurveNormExpr(0);
Expr ny = new CurveNormExpr(1);
Expr nz = new CurveNormExpr(2);
Expr eqnC = Integral(OnGeometry,
    nx*(dx*u)*v+ny*(dy*u)*v+nz*(dz*u)*v,quad_s,geometrySInt);
```

For a polygon geometry representation, the user code is shown in Code 18. The main difference to the previous code is the usage of the *PolygonQuadrature* quadrature class. This is a wrapper class for a quadrature rule that provides the quadrature rule for a line segment of the polygon, and implements the presented boundary integration for polygons. The user has the option to specify the maximum number of line segments within a cell by calling $PolygonQuadrature :: setNrMaxLinePerCell(10)$. By default this is set to six[14] and it is important that the user sets it to the correct maximal value. If, within a cell, there are more line segments than the specified one, it results in the

[14] Assuming that the mesh and the polygon have similar resolutions.

breakdown of the simulation. For 3D, the user code is presented in Code 19, where, for the same boundary integration, one additional component nz of the normal vector is needed. In addition, a special quadrature class $SurfQuadrature$ needs to be used that wraps a quadrature class and uses its quadrature points for each triangle in the boundary surface.

7.4. Fluid Flow Benchmark Results

In the last section of this chapter, we demonstrate the capabilities of the presented cut-cell and boundary integral methods by applying them within the Sundance PDE toolbox to Nitsche's method for the Navier-Stokes equations. (We used these capabilities in the boundary error convergence analysis of the Poisson in Tab. 4.1, where we achieved second order accuracy on the boundary.)

Nitsche's method is a consistent method to impose a given Dirichlet boundary condition. In the fluid simulation, this is the most common type of boundary conditions. One way to verify the method and its presented implementation is to compute the 2D and 3D benchmark scenarios in [80]. These scenarios describe a channel flow, where the upwind part of the channel contains an obstacle. The benchmark values are computed on the boundary of this obstacle and are mainly represented by lift and drag coefficients. Since these values are computed on the boundary, the consistent imposition of the BC is crucial. Our tests showed, that an inconsistent method such as the penalty type methods result in similar flow fields as the ones with Nitsche's method, but the boundary benchmark values, especially the lift and drag coefficients are distorted. With the outlook to fluid-structure interaction, the lift and drag forces on the boundary are crucial for a consistent coupling of the fluid and structure. For more details on the benchmark scenario descriptions in 2D and 3D, transient and stationary, we refer to [80].

Nitsche's method for the Navier-Stokes-equation has been introduced in this thesis by Equation (4.23) in Chapter 4. Here, we restate this equation in the same form in order to illustrate the integrated method's usability. The velocity unknown and test functions are denoted by v and ψ respectively, whereas p and ξ denote the pressure unknown and test functions. Ω represents the computational domain and Γ is the boundary, where the g Dirichlet BC for the velocity is imposed on. On the discrete spaces $V_h \times Z_h$, the equation has the form

$$a(u,\phi) := \nu\,(\nabla v, \nabla\psi)_\Omega + ((v\cdot\nabla)v,\psi)_\Omega - (p, \nabla\cdot\psi)_\Omega + (\nabla\cdot v, \xi)_\Omega\,,$$

$$c(u,\psi) := -\nu\langle\partial_n v,\psi\rangle_\Gamma + \langle pn,\psi\rangle_\Gamma\ ,\ \hat{c}(v,\phi) := -\nu\langle\partial_n\psi, v\rangle_\Gamma - \langle\xi n, v\rangle_\Gamma,$$

with $u = (u, p)$ and $\phi = (\psi, \xi)$.

$$\begin{aligned} &a(u_h, \phi_h) + c(u_h, \psi_h) + \hat{c}(v_h, \phi_h) + \nu \frac{\gamma_1}{h} \langle v_h, \psi_h \rangle_\Gamma + \frac{\gamma_2}{h} \langle v_h \cdot n, \psi_h \cdot n \rangle_\Gamma \\ &= (f, \psi_h)_\Omega + \hat{c}(g, \phi_h) + \nu \frac{\gamma_1}{h} \langle g, \psi_h \rangle_\Gamma + \frac{\gamma_2}{h} \langle g \cdot n, \psi_h \cdot n \rangle_\Gamma \;\; \forall\, \phi_h \in V_h \times Z_h. \end{aligned} \quad (4.23)$$

In Equation (4.23), the volume and boundary integrals are separated. Since the terms in (4.23) are computed on a Cartesian mesh with immersed boundaries, the developed cut-cell and boundary integral methods are deployed within Sundance. The implementation of complex formulas, such as Equation (4.23), is surprisingly easy within the Sundance toolbox with the added capabilities. The user code for Nitsche's method is shown in Appendix A.4 for the benchmark scenarios.

7.4.1. 2D Benchmark Results

Stationary Case

In a first step, to demonstrate the capability of Nitsche's method in fluid mechanics, we set up a stationary scenario, called 2D-1 that is described in [80]. The results with the presented methods have been published in our previous work [19]. This stationary benchmark scenario consists of a channel flow with $Re = 20$ and a cylinder obstacle in the middle as shown in Fig. 7.20. We discretized the cylinder by a polygon containing 63 line segments. For the spatial discretization, we use Q_2Q_1 elements that are quadratic for the velocities and bilinear for the pressure. In the computations, as described in the previous section, we use the cut-cell and boundary integrations for the polygon. The stationary flow field is presented in Fig. 7.20 together with the adaptive Cartesian mesh. We use the feature of the developed Cartesian mesh within Sundance to deactivate the cells that are completely within Ω_F. The pressure field is smooth also at the boundaries, as it can be seen in Fig. 7.20. A penalty method would enforce the Dirichlet BC for velocities on Γ, but it would trigger various artifacts in the pressure field in the vicinity of Γ that would lead to incorrect lift and drag forces on this boundary.

We compute the benchmark lift and drag coefficients. The underlying lift and drag forces can be computed in two different ways. The first approach is the classic one that uses the curve integral over the boundary Γ of the cylinder [80],

$$F_\phi = \oint_\Gamma \phi \cdot \sigma(u) \cdot \mathbf{n} \, dS(x), \quad (7.7)$$

where $\sigma(u) = 2\nu\varepsilon(v) - pI$ is the stress tensor, $\varepsilon(v) = \frac{1}{2}(\nabla v + \nabla v^T)$ is the strain tensor, and ϕ is a unit vector pointing in force direction. This way, (7.7) with $\phi = (1, 0)^T$ becomes the drag force integral and with $\phi = (0, 1)^T$ the lift force integral is computed, since the inflow velocity of the channel is only in the x-direction. Further, $\mathbf{n}$ denotes the

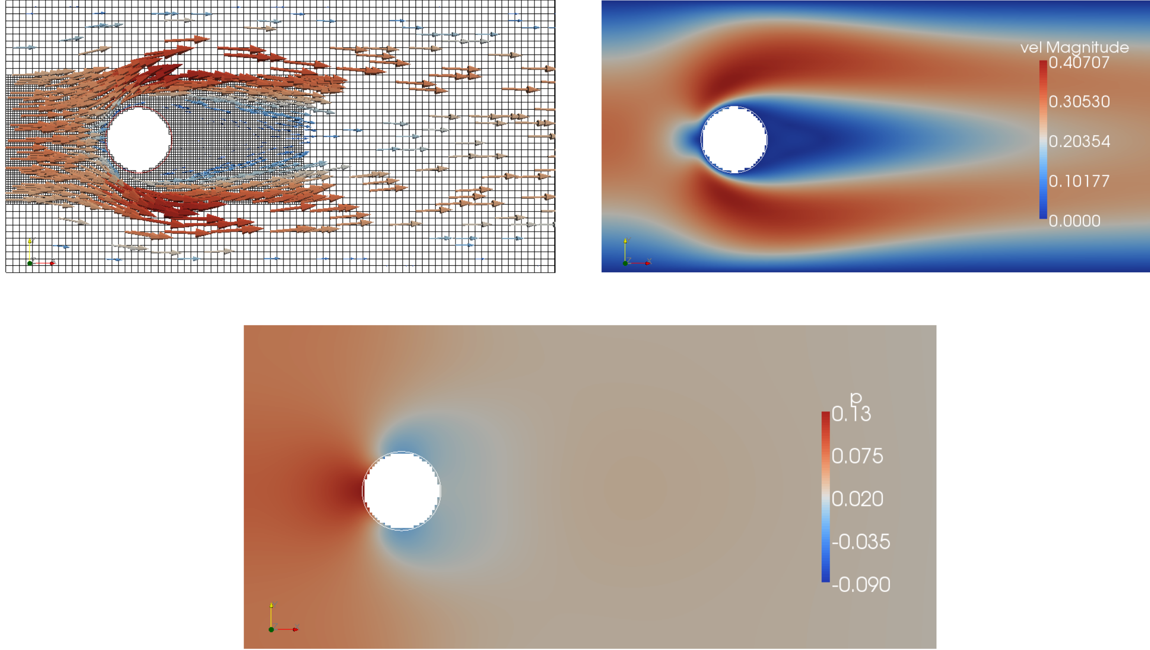

Figure 7.20.: Illustration of the adaptive Cartesian mesh (top left) with 221×42 cells and refined in a rectangular area around the cylinder. The Dirichlet zero BC is imposed on the polygon (top right). The pressure field is smooth also near the boundary (bottom).

outward unit normal vector of the cylinder. The second approach is described in [36]. It transforms the integral (7.7) into a volume integral. This approach proves to be more stable, since it is not influenced by local errors on the boundary:

$$F_{\phi} = -\int_{\Omega} \left[((v \cdot \nabla)v - f) \cdot \Phi - p\nabla \cdot \Phi + 2\nu\varepsilon(v) : \varepsilon(\Phi) \right] dx,$$

where Φ is a smooth extension of the vector ϕ such that $\Phi|_{\Gamma} = \phi$ and $\Phi|_{\partial\Omega \setminus \Gamma} = 0$. This approach can only be used for the total force integrals that can be limitedly applied for fluid-structure interaction. Therefore, our main focus is on the first approach.

One important aspect of Nitsche's method is the choice of the penalty parameters γ_1 and γ_2 that weigh the stabilization terms. We are not aware of any publication that estimates these penalty coefficients for the Navier-Stokes equations. Therefore, we calculated this scenario with several parameter values and resolutions. We made two different convergence analysis for $\gamma_1 = \gamma_2 = 10^3$ and for $\gamma_1 = \gamma_2 = 10^4$. For each setup, we computed the lift and drag coefficients with the two described approaches. The resulting values are presented in Tab. 7.1. For the first case ($\gamma_1 = \gamma_2 = 10^3$), the volume integrals converge to the reference interval, the lift value of the curve integral for the finest resolution still has around three percent relative error. In the second case (with $\gamma_1 = \gamma_2 = 10^4$), both

$\gamma_1 = \gamma_2 = 10^3$	DragV	LiftV	DragC	LiftC
111×21	5.57919	0.012239	5.56063	0.0152477
221×42	5.57935	0.010597	5.57857	0.0098645
441×81	5.57935	0.010622	5.58070	0.0102580
Reference values [16]	5.579535	0.0106189	5.579535	0.0106189
Reference intervals [80]	$5.57 - 5.59$	$0.0104 - 0.0110$	$5.57 - 5.59$	$0.0104 - 0.0110$
$\gamma_1 = \gamma_2 = 10^4$	DragV	LiftV	DragC	LiftC
111×21	5.57936	0.0120811	5.57367	0.0171845
221×42	5.57936	0.0105961	5.57805	0.0105291
441×81	5.57936	0.0106214	5.57801	0.0110584
Reference values [16]	5.579535	0.0106189	5.579535	0.0106189
Reference intervals [80]	$5.57 - 5.59$	$0.0104 - 0.0110$	$5.57 - 5.59$	$0.0104 - 0.0110$

Table 7.1.: Results of the 2D-1 benchmark computations. The values in the columns "DragV" and "LiftV" are the drag and lift coefficients computed with the volume integrals on Ω. The values in the columns "DragC" and "LiftC' are the drag and lift values computed by the direct curve integrals on the boundary Γ of the cylinder.

approaches show a more stable convergence and result in benchmark values, which are either inside or near the reference interval specified in [80]. We further observe that for higher penalty coefficients the volume integral's results are almost identical to the previous case, which also show that the volume integrals are a more stable approach to compute the global lift and drag forces. The presented results were computed on a desktop quad-code Intel i7 2.9GHz machine. Even for the highest resolution, the runtime was only a couple of minutes. The Sundance code for this stationary scenario can be found in Appendix A.4.

Transient Case

In the following, we compute a 2D transient benchmark scenario, called 2D-2 in [80]. In comparison to the previous scenario, the inflow velocity is increased ($Re = 100$), such that Karman vortices are created during this simulation. The governing equation is the same (Equation (4.23)), except that the time derivative $\frac{v_{n+1}-v_n}{\Delta t}\psi$ is added. The resulting equation's time discretization is handled by the Crank-Nicolson scheme. In this transient case, the velocity field is fluctuating as the different snapshots in Fig. 7.21 show. In the same manner, the resulting drift and lift coefficients are time dependent and also show an oscillating behavior (Fig. 7.22). In this case, the benchmark values are the maximum values of the drag and lift coefficients and the Strouhal number that represents the oscillation frequency of the drag and lift forces. For the transient computations, we choose a lower penalty number since the Dirichlet BC is enforced in each time step continuously. In Tab. 7.2, we present the results and compare them to the benchmark intervals from [80]. The discrete time step for all the simulations was set

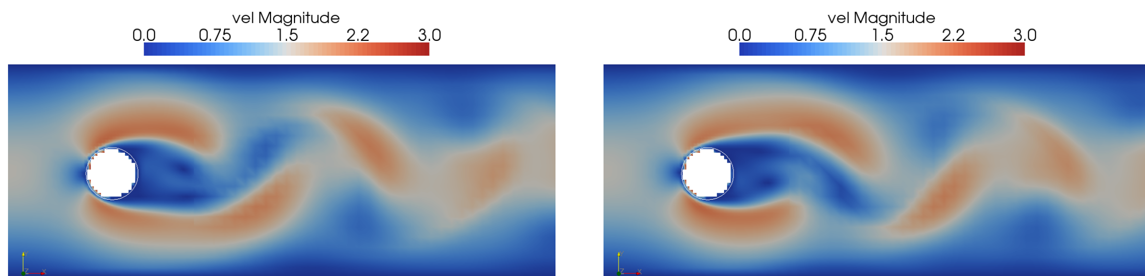

Figure 7.21.: The velocity field of the transient scenario plotted at different time steps.

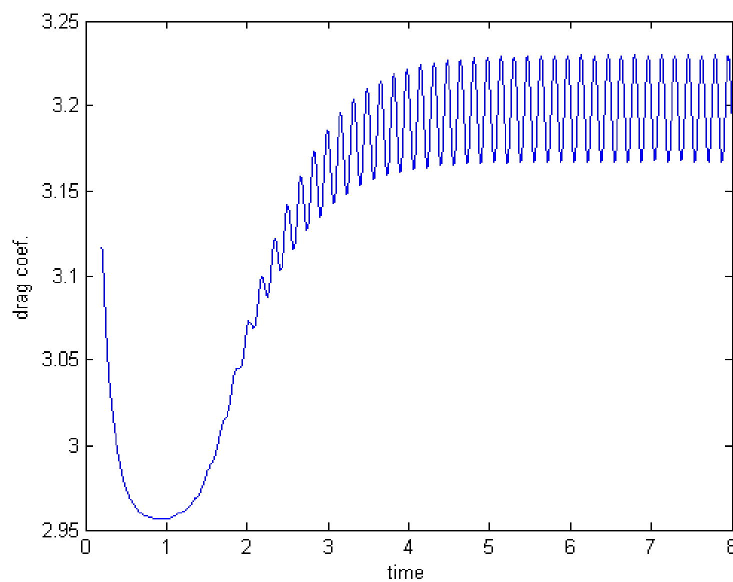

Figure 7.22.: Time-dependent drag coefficient of the 2D-2 transient benchmark scenario. Time is given in seconds.

to 10^{-3}. Since, for a complete simulation, at least 6000 time steps were necessary, we used a lower resolution in comparison to the stationary case. Even with such lower spatial resolution, the maximal lift and drag coefficients are either inside or near[15] the benchmark intervals [80]. The computed Strouhal number also matches the benchmark value, and the time variant behavior of the simulated system proved to be accurate. The Sundance code for this transient simulation can be found in Appendix A.4.

[15]less than 2% relative error

$\gamma_1 = \gamma_2 = 2 \cdot 10^2$	DragV	LiftV	DragC	LiftC	St
111×21	3.2295	0.9841	3.2188	0.9861	0.3020
150×26	3.2300	0.9855	3.2252	0.9880	0.3004
Reference intervals [80]	3.22-3.24	0.99-1.01	3.22-3.24	0.99-1.01	0.295-0.305

Table 7.2.: Results of the 2D transient simulation. $DragV$ and $LiftV$ represent the maximum values of the drag and lift coefficients over the time computed by the volume integrals, whereas $DragC$ and $LiftC$ are the drag and lift values computed by the curve integrals on Γ.

7.4.2. 3D Benchmark Results

In 3D, we compute a stationary benchmark scenario from [80] called 3D-1Z. The scenario is a 3D channel flow (with $Re = 20$), where, in the middle of the channel, a cylinder obstacle is placed as illustrated in Fig. 7.23. Since the cylinder is slightly in the lower part of the channel, similar to the stationary 2D scenario, a small positive lift value is expected. For more details about the scenario, we refer to [80]. The cylinder obstacle is modeled by a surface with 40 triangles. Nitsche's method in 3D has the same form as presented in (4.23). In 3D, we compute the lift and drag forces by the surface integral (7.7), where the vector ϕ has three components. $\phi = (1, 0, 0)^T$ is used for the drag integration, whereas $\phi = (0, 1, 0)^T$ results in the lift force integration. In order to fulfill the inf-sup condition (see Chapter 3), we choose the Q_2 basis for the velocities and Q_1 for the pressure. The resulting Q_2Q_1 element in 3D requires 89 local DoFs. Therefore, we also employed a stabilized Q_1Q_1 element, described in Chapter 3. Such a Q_1Q_1 stabilized element needs only 32 local DoFs, and thus, allows for more mesh refinement in 3D compared to the Q_2Q_1 element.

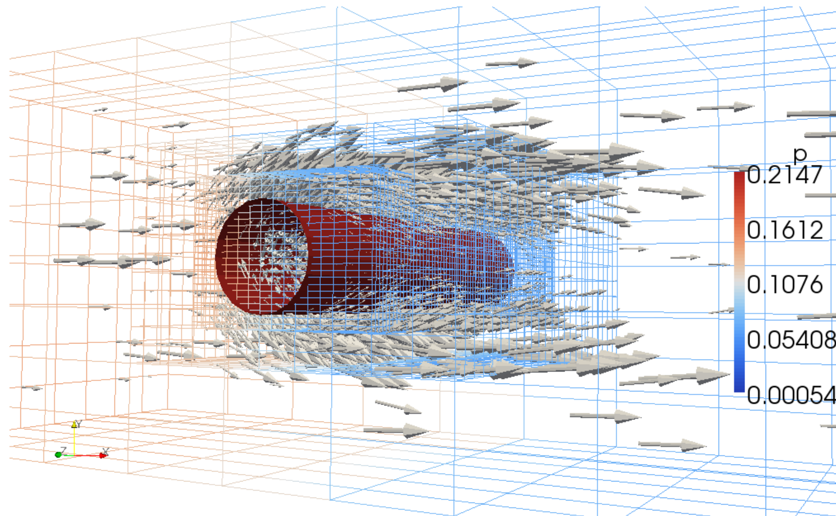

Figure 7.23.: Stationary 3D benchmark, computed with a Q_1Q_1 element. The obstacle in the channel flow is represented by a triangulated surface.

We computed the 3D benchmark scenario with both types of elements. The results are shown in Tab. 7.3. With the stabilized Q_1Q_1 elements, we were able to achieve considerably higher spatial resolution, than with the Q_2Q_1 elements. The limiting factor is not just the total number of DoFs, but also the assembly of the system matrix that becomes more here compared to 2D. The convergence results in Tab. 7.3 show that with

Q_1Q_1, $\gamma_1 = \gamma_2 = 10^2$	DragC	LiftC	#Cells
$40 \times 13 \times 13$, $l = 1$	6.100	0.021	18916
$20 \times 7 \times 7$, $l = 2$	5.810	0.077	25764
$22 \times 8 \times 8$, $l = 2$	6.053	0.071	32783
Reference intervals [80]	6.05 - 6.25	0.008-0.01	
Q_2Q_1, $\gamma_1 = \gamma_2 = 5 \cdot 10^2$	DragC	LiftC	#Cells
$25 \times 10 \times 10$, $l = 1$	5.539	-0.107	2502
$27 \times 11 \times 11$, $l = 1$	6.000	0.078	7624
$26 \times 12 \times 12$, $l = 1$	6.021	0.432	9113
Reference intervals [80]	6.05 - 6.25	0.008-0.01	

Table 7.3.: Results of the 3D stationary benchmark computations. *DragC* and *LiftC* represent the coefficients computed by the surface integrals. The first table contains the stabilized Q_1Q_1 elements' results, whereas the lower table shows the results with Q_2Q_1 elements. Besides the initial spatial resolution, l represents the number of additional refinement levels around the cylinder obstacle as shown in Fig. 7.23.

the stabilized Q_1Q_1 elements the resulting drag coefficients correspond to the benchmark interval, whereas the lift coefficients miss the reference interval slightly. For the Q_2Q_1 elements, even though only 9113 cells could be computed, the resulting drag coefficient differs from the lower bound of the interval only at the third digit. The lift coefficient has the same sign as the benchmark value, but the error is larger in comparison to the previous results. This is due to the low mesh resolution. From the results in Tab. 7.3, we can conclude that the developed cut-cell and boundary integral methods in 3D applied to Nitsche's method perform well and were capable to compute benchmark values for the Navier-Stokes equations. The Sundance code for this 3D stationary scenario can be found in Appendix A.4.

8. Fluid-Structure Interaction with Nitsche's Method

In this chapter, we extend the approach from the previous chapter to fluid-structure interaction problems. Since Nitsche's method with our IB implementation has been verified in the last chapter for stationary geometries, the first step is to extend our approach also to moving boundaries. The required extensions of Nitsche's method and Sundance are discussed in the first section. The next step towards a FSI simulation is the coupling of the Lagrangian and Eulerian frameworks. It has been shown in Chapter 3 that the fluid flow is modeled in the Eulerian framework, whereas the structure is computed efficiently in the Lagrangian framework.[1] In order to bridge the gap between these two settings, we develop the construct of **twin polygons** and **twin triangular surfaces**. These constructs facilitate the mapping between the Eulerian and Lagrangian frameworks. A similar approach has also been employed in the coupling research software preCICE[2] [25, 26] that can couple two solvers in their original Eulerian or Lagrangian settings. We use these developed features in Sundance to simulate various FSI scenarios, where both the structure and the fluid equation are set up in Sundance, but the coupling is done with implicit or explicit partitioned approach. Our approach to use Nitsche's method in an IB context for FSI problems appears to be unique in the literature. Therefore, we compute several 2D benchmark scenarios in order to verify our approach. The presented simulation results in 2D and 3D demonstrate the true potentials of Nitsche's method for FSI applications, since they allow for the usage of a fixed Cartesian mesh for the fluid, where the Dirichlet BCs are imposed on IBs.

8.1. Moving Geometries with Nitsche's Method in 2D and 3D

Transient FSI scenarios require the fluid solver to handle moving boundaries in the flow field. In order to show the challenges with moving boundaries, we consider the 2D illustration in Fig. 8.1. The boundary is represented by a circle that is moving in the flow direction. We assume that we have a solution of the velocity field at time t_1 and

[1] There are approaches to compute the structure in an Eulerian setting as well [31].
[2] http://www5.in.tum.de/wiki/index.php/PreCICE_Webpage

the question is how to compute the solution at t_2. We denote accordingly the fictitious domain of the two time steps as Ω_{F1} and Ω_{F2}, and the computational domains as Ω_1 and Ω_2, which are denoted by Ω in Fig. 8.1. The problem arises at t_2, when the domain $\Omega_{F1} \setminus \Omega_{F2}$ becomes part of Ω_2. In this domain the velocity values at t_1 don't fulfill the Navier-Stokes equations, since in the previous time step they were part of the fictitious domain Ω_{F1}. We write the transient Navier-Stokes equations with Nitsche's method from (4.23) and (4.23) in the simplified form

$$\left(\frac{v_{t_2} - v_{t_1}}{\Delta t}\right)\psi = \left(a\, L\left(v_{t_2}, \psi\right) + (1-a)\, L\left(v_{t_1}, \psi\right) + C\left(v_{t_2}, p, \psi, \xi\right)\right), \qquad (8.1)$$

where L represents the diffusion and the convection terms in their weak forms. C contains the continuity and pressure gradient terms in the weak form and a the parameter that defines the time integration. In the previous chapter, we set $a = 0.5$ and it resulted the Crank-Nicolson scheme with (v_{t_2}, p) as unknowns. Since in $\Omega_{F1} \setminus \Omega_{F2}$ the values of v_{t_1} do not fulfill the Navier-Stokes equations, we want to limit the impact of these values. Therefore, we choose a full implicit scheme $a = 1$, such that only the time derivative term contains v_{t_1}. In addition to the full implicit method, we solve on each fictitious

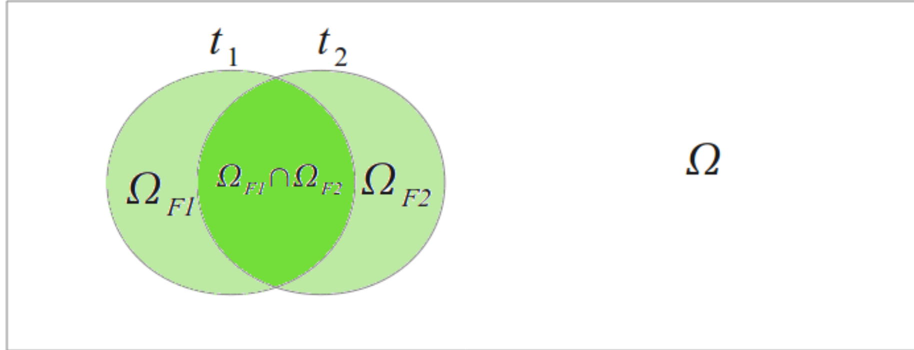

Figure 8.1.: Illustration of the moving geometry problematic. The two circles represent the moving geometry's position at two consecutive time steps t_1 and t_2. The fictitious domains Ω_{F1} and Ω_{F2} are marked accordingly on the figure.

domain Ω_{Fi} a Poisson equation weighted by 10^{-6}, to make sure that the velocities v_{t_1} in the neighborhood of the moved boundary do not take extreme values. We also recall that the velocities with Nitsche's method near the boundary have reasonable values. Therefore, if we consider only small movement of the boundary between t_1 and t_2, the error introduced by v_1 in $\Omega_{F1} \setminus \Omega_{F2}$ is limited. With respect to the FSI applications, it is crucial that the moving boundaries are implemented consistently such that the resulting coupling forces are not altered. This will be proved for transient scenarios in later sections of this chapter.

Implementational Aspects in Sundance

A moving boundary also represents additional implementational requirements for the Sundance toolbox, where we developed the presented cut-cell and boundary integral methods. In order to be efficient, we store the precomputed quadrature weights and points in the mesh object. In addition, Sundance caches the results from the cell filters for efficient mesh assembly of transient problems. Once the boundary has been moved, all these information need to be flushed and the recomputation of them should be triggered. This has been implemented by the additional function $problem.reAssembleProblem()$, where the *problem* can be a linear or non-linear problem object. After calling this function, all the cached information are deleted, such that the cell filters are evaluated for each cell, the special quadrature weights and points are recomputed, and the system matrix is assembled newly.

In addition, we also had to facilitate the movement of the boundary. For these cases, we only consider the polygon and triangular surface boundary representation and we neglect the analytical boundary representation. Both representations offer the direct access to the points that form them. This is illustrated in Code 20, where we consider a polygon with two defined nodal functions on it, which represent the velocities. By using the velocity of each point on the polygon, we update the position as $\mathbf{x} = \mathbf{x} + \Delta t\, \mathbf{v}$. The

Code 20 Sundance code to illustrate the update in the polygon's position. We assume that the two scalar fields, representing the point-wise velocities, have been set previously. Once the position of the polygon has been changed, it is crucial that the *.update*() function of the polygon and the $.reAssembleProblem()$ function of the problem object are called.

```
ParametrizedCurve polygon = new Polygon("polygon.txt",1,1e-8);
...
double dt = 1e-3;
Array<Point>& pnt_polygon = polygon.getControlPoints();
Array<double>& velX = polygon.getScalarFieldValues(0);
Array<double>& velY = polygon.getScalarFieldValues(1);
for (int p = 0 ; p < pnt_polygon.size() ; p++ ){
      pnt_polygon[p][0] = pnt_polygon[p][0] + dt*velX[p];
      pnt_polygon[p][1] = pnt_polygon[p][1] + dt*velY[p];
}
// let the polygon known that its positions have been changed.
polygon.update();
// trigger reassemble of the problem
problem.reAssembleProblem();
```

position $\mathbf{x}$ of the polygon is accessed by the $.getControlPoints()$ function of the polygon. Accordingly, the $TriangleSurf3D$ class also implements this method, and Code 20 could

be applied in a similar manner to a $TriangleSurf3D$ object. The velocities $\mathbf{v}$ of the polygon are accessed also directly by the $getScalarFieldValues(i)$ function. Once the polygon's position has been updated, the $.update()$ function is called, such that the internal information[3] of the polygon are recomputed. Finally, the $.reAssembleProblem()$ method of the problem object is called, such that all cached results are flushed and a reassembly of the system matrix is triggered. For 3D and for a $TriangleSurf3D$ object, the Sundance code would have the same structure and the same functions of the geometry would be called. Such 3D Sundance codes are listed in the Appendix A.5 for FSI simulations and will be discussed later in this chapter.

8.2. Partitioned Fluid-Structure Interaction

The mathematical formulation of an FSI problem has been already discussed in Chapter 3 for both stationary and transient cases. Therefore, we consider in this section only the problem to couple the two different equations according to the coupling equations described in Chapter 3. Since the fluid is modeled in an Eulerian setting and the structure is naturally simulated in a Lagrangian setting, one way to set up the FSI problem is to solve these equations separately, and couple them only through the right-hand side of the systems. This approach is called **partitioned** coupling, since two separate systems are solved, which are coupled iteratively. In contrast to this, the **monolithic** coupling assembles the two equations into one system, where the coupling is done implicitly through matrix entries. This also involves the formulation of the fluid and the structure in the same setting. One solution for monolithic coupling is the formulation of the fluid by the Arbitrary Lagrangian Eulerian (ALE) approach [45] or by formulating the structure in a pure Eulerian setting [31]. For more details on monolithic approaches, we refer to [30, 35, 90].

In the following, we discuss the algorithmic aspects of the partitioned approach to couple the fluid and the structure systems for stationary and transient cases in order to consistently simulate the coupled system. We only highlight the coupling methods employed later in the FSI simulations, that were already used previously in [25]. For more insights on the partitioned approach, we refer to [30, 35, 22].

8.2.1. Stationary FSI

In the first step, we consider the stationary partitioned FSI coupling. In Chapter 3, this case was already introduced. This implies that there are only two quantities that are coupled, since the velocity of the structure is by definition zero. These two quantities

[3] e.g., assign to each polygon point one cell LID, where it is located. This information is needed by the polygon's cut-cell method.

are the forces (or stress vectors) and the displacements, and these quantities are passed from one solver to the other as illustrated in Fig. 8.2. According to this illustration,

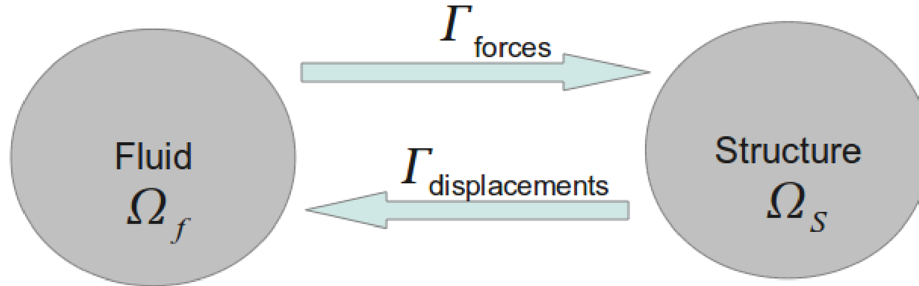

Figure 8.2.: Illustration of the stationary partitioned coupling. The quantities to couple between the two solvers are the displacements and the forces.

one iteration of coupling starts with solving the fluid system for an initial position u_0 of the structure. The resulting forces are passed to the structure solver that also returns the new position u_1 of the structure. This results in an iterative method where from a given u_i a new displacement u_{i+1} results, and the process should be repeated until $\|u_{i+1} - u_i\| < \varepsilon$. This process is presented in Alg. 1, where the update of the actual boundary displacement is under-relaxed with the factor w.

Algorithm 1 Algorithm for partitioned static FSI coupling. w is an under-relaxation factor for the coupling.

```
u_0 = 0
i = 0
do
    solve Fluid with u_i as boundary position
    extract forces on Γ
    solve Structure with the forces (or stresses) as BC
    extract displacements d_{i+1}
    u_{i+1} = u_i + w(d_{i+1} - u_i)
    i = i + 1
until ||u_i - u_{i-1}|| < ε
```

8.2.2. Partitioned and Transient FSI with Explicit and Implicit Time Coupling

There are two main approaches for the simulation of transient FSI scenarios with a partitioned coupling. The first approach is represented by **explicit** methods that solve

the fluid and the structure in time and couple only once the forces, displacements and velocities, at one discrete time. One such coupling scheme is the explicit serial staggered scheme illustrated in Fig. 8.3, where $\underline{\sigma}_f \cdot \mathbf{n}$ represents the stress vector[4] on the coupling interface, u_S and v_S represent the displacements and the velocities respectively. This

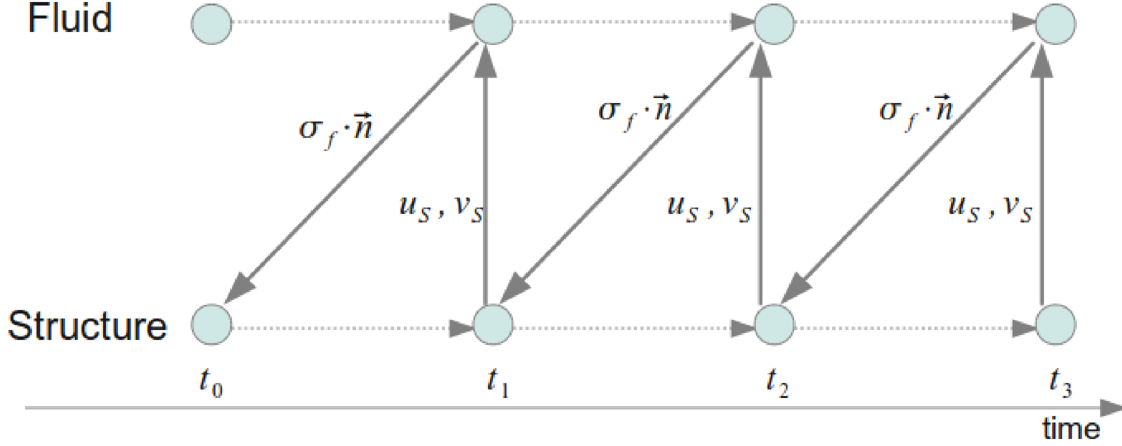

Figure 8.3.: Explicit coupling method with the serial staggered scheme.

staggered coupling is also illustrated in Alg. 2. We employ this explicit coupling scheme for some of the transient FSI scenarios, where this coupling scheme is stable. The criterion for the stability lies on the one hand mainly in the density ratio of the fluid and structure materials [30] and on the other hand in the stiffness of the structure. Unfortunately, all the benchmark scenarios that we computed, lead to instability of the explicit coupling scheme. In all these cases, the explicit scheme does not accurately capture the so-called added mass of the structure induced by the fluid resistance [30].

Algorithm 2 Algorithm for the serial staggered scheme for explicit partitioned coupling. u_S and v_S denotes the structure's displacements and velocities, whereas $\underline{\sigma}_f$ represents the stress tensor of the fluid.

```
u_0 = 0
i = 0
for t=0 till T with step Δt do
    solve Fluid for Δt with u_S and v_S as boundary conditions
    extract the stress vector σ_f · n on Γ
    solve Structure for Δt with σ_f · n as boundary condition
    extract u_S and v_S
end for
```

For these reasons, **implicit** coupling schemes are required, which ensure that after each time step the coupled system is converged. Similar to [30], we consider the coupled

[4] The force vector is computed as $\mathbf{F} = \oint_\Gamma \underline{\sigma}_f \cdot \mathbf{n} dc$.

problem as a function in terms of displacements u such that $u^t_{i+1} = Sr \circ Fl\,(u^t_i)$. Sr represents the structure solver whereas Fl denotes the fluid solver. The goal is to find a displacement u^t for a given time step t where the equation $\|u^t - Sr \circ Fl\,(u^t)\| < \varepsilon$ holds. In order to find this equilibrium for each time step t, several iterations are required within each time step. This implicit coupling is illustrated in Fig. 8.4. This cycling is illustrated in Fig. 8.4 in three time steps. There are several ways to determine the

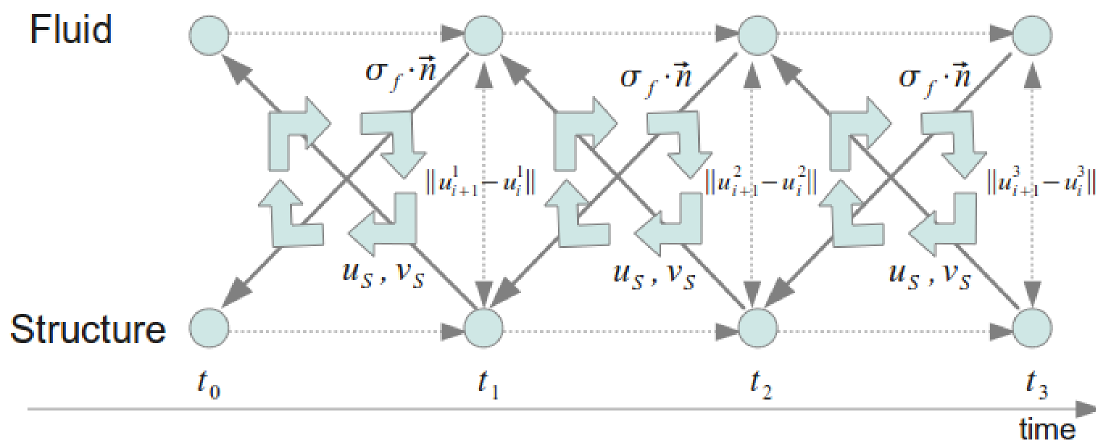

Figure 8.4.: Implicit coupling method. At each time step, the solution of $\|u^i - Sr \circ Fl\,(u^i)\| < \varepsilon$ needs to be determined, and only then can be stepped forward in time.

solution of the nonlinear equation $u^t - Sr \circ Fl\,(u^t) \approx 0$. Since this equation is non-linear, a quasi-Newton methods might be a good choice. [30] describes several quasi-Newton methods such as IQN-ILS and IQBN-LS, and it also shows that such methods potentially require a smaller number of iterations compared to other conventional methods, such as Aitken or interface-GMRES. The coupling could be further accelerated using a multi-level coupling approach [30].

In this thesis, we used the Aitken iteration method that avoids the usage of any gradient dependent information. Instead, it uses a dynamic under-relaxation factor w that was first used for FSI in [55]. This factor is used during the cycling within one time step to determine the solution u^{k+1} in Alg. 3. This implicit coupling with the Aitken iterations was implemented directly in the Sundance C++ user code and we did not integrate this method into Sundance, as a toolbox feature.

8.2.3. Implementational Requirements in Sundance

The last missing feature for a transient FSI simulation is the mapping between the Lagrangian and Eulerian settings. Note that due to Nitsche's method used in the flow solver, this mapping corresponds to the identity in a numerical sense but technically

Algorithm 3 Aitken iterations for the cycling of one time step for FSI implicit coupling.

$k = 0$
$\hat{u}^1 = Sr \circ Fl(u^0)$
$r^0 = \hat{u}^1 - u^0$
while $\left\|r^k\right\| < \varepsilon_0$ do
  if k=0 then
    $w^0 = sign(w^n)min(|w^n|, w^{max})$
  else
    $$w^k = -w^{k-1}\frac{(r^{k-1})^T(r^k - r^{k-1})}{(r^k - r^{k-1})^T(r^k - r^{k-1})}$$
  end if
  $u^{k+1} = u^k + w^k r^k$
  $k = k + 1$
  $\hat{u}^{k+1} = Sr \circ Fl(u^k)$
  $r^k = \hat{u}^{k+1} - u^k$
end while

requires connecting values at polygonal nodes in the Eulerian (fluid) setting with those in the Lagrangian (structure) setting. This task has to be done by the boundary representation. In coupling softwares such as preCICE[5] [25, 26], this mapping is also done by the boundary mesh. We use for this purpose only the polygon representation in 2D and the triangular surface representation in 3D. To demonstrate the idea of the mapping, we consider the twin polygons in Fig. 8.5. Our idea for data mapping between the Eulerian and Lagrangian setting is to create two identical polygons, one for each setting. Between the points of the polygons, there is a bijective mapping that is illustrated by the dotted lines in Fig. 8.5. This means, if the values are set in the Eulerian framework, then along these values, the nodal values can be copied to the Lagrangian framework and vice versa. The same idea has been implemented for the triangular surface in 3D. In terms of software features, this implies that a polygon can have a **twin polygon** and has the same number of nodal functions defined. Once the values of a nodal function are set on the polygon, these nodal values should be copied to the twin polygon's corresponding function in an automatic manner. This feature facilitates the automatic mapping between the two equations discretized in different settings.

In the parallel case, the mesh is decomposed among the processors, such that no single processor can set all the nodal values of the polygon. In these cases, one processor sets the nodal values, which are covered by its local mesh. After this with an *allReduce* MPI command, the nodal values are centralized and redistributed among the processors, such that each processor has the complete nodal values of the polygon. This is important also for the twin polygons, since their nodal values will be also updated consequently. The described approach has been implemented also for triangular surfaces in 3D. The

[5] http://www5.in.tum.de/wiki/index.php/PreCICE_Webpage

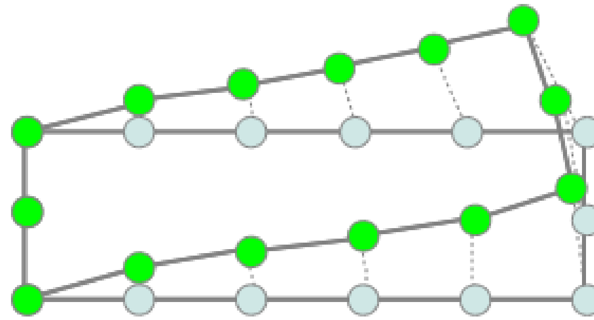

Figure 8.5.: Illustration of twin polygons. The main idea is to clone one polygon, such that there is a bijective mapping between the two polygons (marked with blue and green). One of the polygons is in Eulerian setting (green) and the twin polygon (blue) is in Lagrangian setting.

Sundance code to create a twin polygon is shown in Code 21. There, we also show, how the nodal values of a polygon and the corresponding twin polygon's values are set. This code is usable in this form not just in the sequential case but also in parallel simulations.

Code 21 Sundance code for twin polygon creation. In the last two lines, we set a nodal function of the polygon with the given expression, then the twin polygon's values are set accordingly.

```
Polygon2D* polygP = new Polygon("polygon.txt",1,1e-8);
ParametrizedCurve curve = polygP;
curve.addNewScalarField( "V" , 0.0 );
curve.addNewScalarField( "U" , 0.0 );
ParametrizedCurve curve_twin=polyg->createTwinPolygon(0,0,1,1);
...
curve.setSpaceValues( VInt , 0 );
curve_twin.setSpaceValues( YInt , 1 );
```

8.3. 2D Results

In order to verify our approach for FSI simulation using Nitsche's method with the developed cut-cell and boundary integral methods, we consider the stationary and transient 2D benchmark scenarios with incompressible flow from [46]. These scenarios describe an

elastic bar that is attached to a cylinder and is subject to fluid forces. All three benchmark scenarios that we compute in this section have the same setting. This setting is presented in Fig. 8.6. On the top and bottom walls, we impose a no-slip BC. The inflow is defined by a parabolic inflow Dirichlet BC in the x-direction, where the mean velocity $\bar{v}_f$ is specified. On this boundary the y-component of the velocity is set to zero and an outflow BC is used on the right side of the channel. Besides the mean velocity $\bar{v}_f$, the characteristic parameters for each scenario are the density ρ_s and the Young modulus of the structure E. The density and the kinematic viscosity of the fluid and Poisson ratio are constant for all scenarios $\rho^f = 10^3 \frac{kg}{m^3}$, $\nu_f = 10^{-3} \frac{m^2}{s}$, and $\nu_s = 0.4$. For more details on these scenarios' description and parameters, we refer to [46]. In the first step,

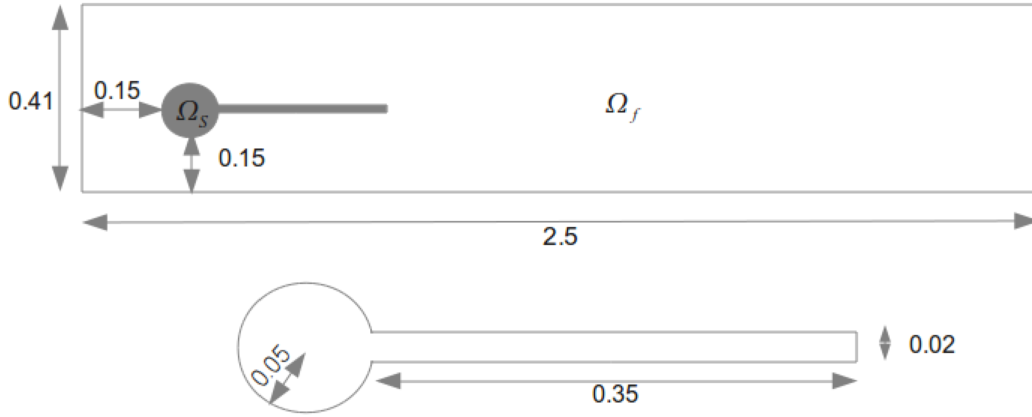

Figure 8.6.: Illustration of the benchmark scenario setting. The obstacle is composed of an elastic bar and a fixed cylinder.

we compute the stationary 2D benchmark, called FSI1 [46], where there is no moving boundary, and only the stationary coupling scheme need to be deployed. Finally we show the results of the two transient benchmark scenarios, called FSI2 and FSI3 in [46].

8.3.1. 2D Stationary Results

The FSI1 benchmark scenario contains the described flexible bar obstacle that is attached to a fixed and rigid cylinder. Since this obstacle is placed slightly in the lower part of the channel, a small lift of the bar is expected. The characteristic parameters for the FSI1 are set as $\bar{v}_f = 0.2 \frac{m}{s}$, $\rho_s = 10^3 \frac{kg}{m^3}$, $Re = 20$, and $E = 1.4 \cdot 10^6 \frac{kg}{ms^2}$. In this case, the benchmark values consist of the x- and y-displacements of the middle tip point of the bar (see Fig. 8.7), and the measured total lift and drag forces on the whole boundary.

As described in the previous chapter, we set up the Navier-Stokes equations in Sundance by using Nitsche's method to impose zero Dirichlet BC. The structure equation, as it was described in Chapter 3, is set up in the Lagrangian form in Sundance. Both equations use a 2D Cartesian mesh. For the structure, the Neumann BCs need to be imposed

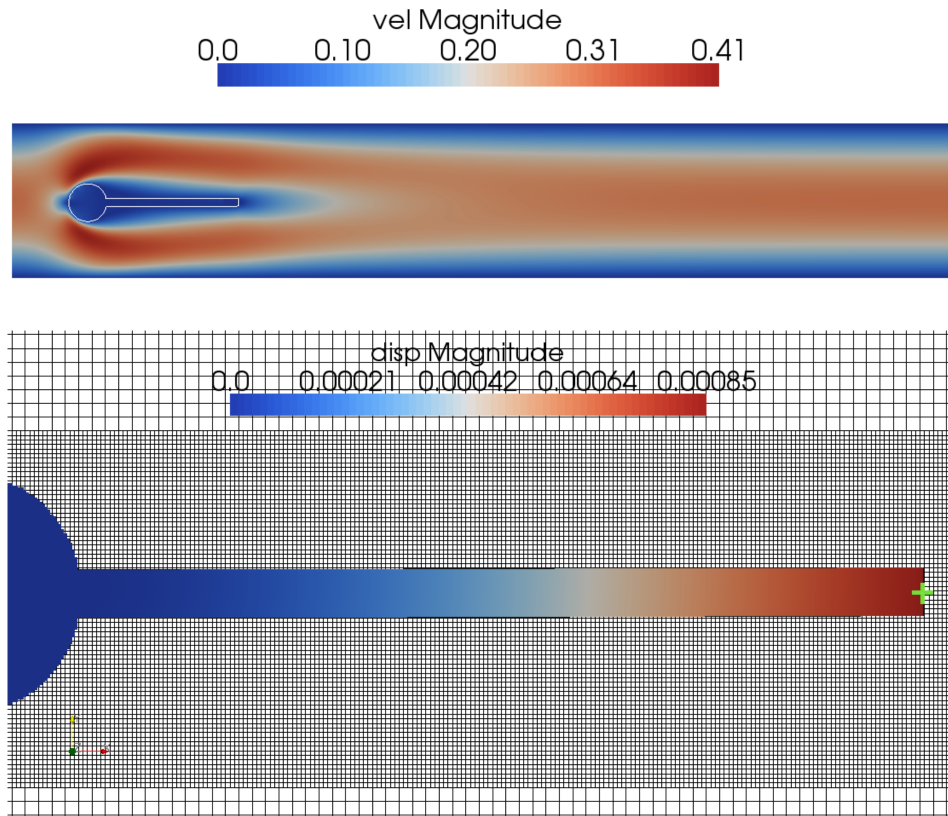

Figure 8.7.: Illustration of the FSI1 scenario. The obstacle is represented by the white polygon line (top). The green cross at the end of the bar shows the reference point, where the displacements are measured.

only on the rectangular bar, since on the cylinder we need to impose zero Dirichlet BCs. We note here, that the Neumann BC imposition is very critical, since it represents the coupling between the two systems. In the Lagrangian setting the polygon representing the boundary of the structure fits exactly the elastic bar boundary. Therefore, there is no need for IB methods on the structure side and the Neumann BC can be imposed exactly. On the cylinder part, the zero Dirichlet BCs are imposed by row replacements in the matrix (see Chapter 5). For complex structure shapes, a given IB method could be employed, such as the Finite Cell Method [72], by using the developed IB capabilities within Sundance. For the incompressible Navier-Stokes, we used Q_2Q_1 elements, whereas the structure was discretized with Q_1 elements.

We use the presented **twin polygon** construct to map the forces from the fluid to the structure and to displace the fluid's polygon according to the structure's displacements. We iterate in Alg. 1 until the update in the displacement is less than $\varepsilon = 10^{-6}$. The under-relaxation factor is set to $w = 0.3$. With such constant under-relaxation, 20 iterations lead to convergence in average. The number of iterations depends not just on w and on the scenario but also on the desired accuracy ε. The benchmark results for different structure and fluid resolution are shown in Tab. 8.1. This shows that the

displacement results for the highest resolution match the benchmark values well, that the total drag and lift forces computed by curve integrals on the polygon also match the benchmark results. The results in Tab. 8.1 verify our approach for stationary FSI scenarios. We further notice, that even for lower resolutions such as $2601 \times 3380 \times 136$ the resulting displacements and boundary forces have at most only 20% relative error, due to the accuracy of Nitsche's method and the developed cut-cell and boundary integrals. The high accuracy on Cartesian meshes without any mesh adjustment or transformation underlines the efficiency of our approach for static FSI scenarios. Parts of the Sundance user code for this static coupling is highlighted in Appendix A.5.

#Fl × #Sr × #Poly	Ax	Ay	Drag	Lift
2601 × 3380 × 136	1.86e-5	0.00123	14.0779	0.820194
36666 × 13370 × 1093	2.21e-5	0.000703	14.1982	0.810081
54104 × 13370 × 1093	2.18e-5	0.000846	14.2236	0.793047
Benchmark values [46]	2.27e-5	0.000821	14.295	0.7638

Table 8.1.: Stationary FSI1 results. The resolution of the three components fluid, structure, and polygon (#Fl × #Sr × #Poly) is specified by the number of cells. Ax and Ay represent the displacements at the reference point. Drag and Lift represent the total drag and lift forces computed on the polygon.

In the following, we consider an additional static FSI scenario, where the resulting relative displacements are considerably higher than in FSI1. In the FSI1 scenario, the displacements were small compared to the structure size, such that a linear model of the structure would result in comparable results. In order to test the static coupling for higher displacements, we set up a different scenario. For this, we use the same channel flow and fluid parameters as in the FSI1 scenario [46]. The main difference is that we increase the inflow velocity from $\bar{v}_f = 0.2\frac{m}{s}$ to $\bar{v}_f = 0.5\frac{m}{s}$, and that the obstacle is represented by a vertical bar that is attached to the lower wall of the channel as shown in Fig. 8.8. Due to the vertical placement of the obstacle in the channel, we expect higher displacements compared to the previous scenario. The deformed structure is shown in Fig. 8.8. Besides the deformed structure, Fig. 8.8 also shows the stress vectors of the equilibrium state. For the coupling method we used a constant under-relaxation parameter $w = 0.31$. Convergence was achieved within 12 iterations with $\varepsilon = 10^{-5}$.

Parallel simulations

As last for the stationary case, we highlight the parallel aspects of the simulation. We consider the previously computed FSI1 scenario. Thanks to the parallel capabilities of Sundance, this scenario can be computed with the same code also in the parallel case. This is one of the true potentials of the developed features in Sundance that even such

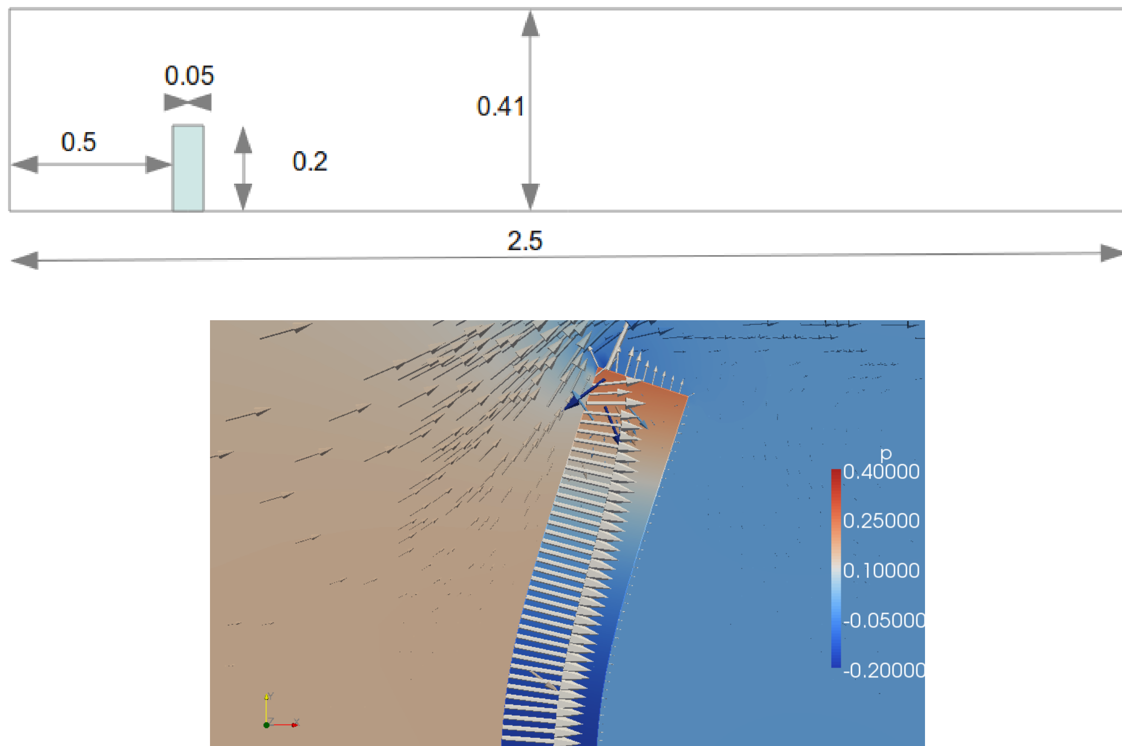

Figure 8.8.: Illustration of the additional static FSI scenario (top) with relatively large displacements. The resulting displacement with the neighboring fluid field is illustrated (bottom) in the equilibrium state, where the white arrows on the polygon represent the stress vectors $\underline{\sigma}_f \mathbf{n}$. The pressure field with velocity vectors in the fluid are also illustrated.

complex multi-physics (FSI) scenarios can be computed sequentially or parallel with the same Sundance user code. The underlying FSI scenario contains two meshes, one for the fluid and the second for the structure. In the partitioned case, only one of the two systems is solved at a given moment of time. Therefore, the parallelization of the FSI computation is restricted to the parallelization of the fluid and the structure systems. The two Cartesian meshes are decomposed based on the Z-curve (see Chapter 6 for details of the implementation). The resulting load-balanced partitions are presented in Fig. 8.9, where the partitions are marked with different colors. Since the cells are refined in a rectangular area around the obstacles, the first two partitions cover a smaller area but the number of cells is similar in all partitions for a load-balanced computation. Consequently, the fluid and the structure systems are computed in parallel on four processors. The twin polygons also have parallel capabilities. In this scenario, the values on the polygon are set by two processors. In general, the processor that sets a nodal value and the one that uses this value might be different. Therefore, the twin polygons synchronize their nodal values after each operation, such that on each processor all the nodal values are available. Even though Sundance has now the required capabilities for parallel FSI simulations,

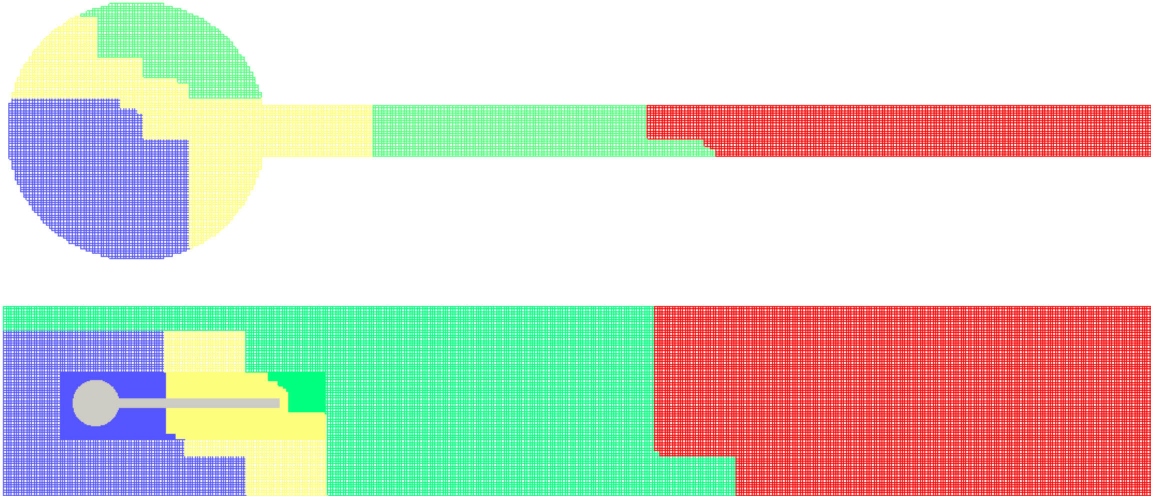

Figure 8.9.: Illustration of the FSI1 parallel simulations with 4 processors. The fluid mesh (bottom) and the structure mesh (top) have been decomposed based on the Z-curve. For the visualization, we used different scales, the white obstacle (bottom) represents the proportional size of the obstacle.

such a computation requires also an efficient parallel solver. For most FSI simulations, we used the direct solver Amesos-KLU[6] that has limited parallel capabilities. For this solver, as expected, the resulting parallel speedup was rather poor. Therefore, for parallel FSI simulations, we used the SuperLU-DIST solver [57] in the linear step of the non-linear NOX solver.[7] For a strong scaling study, we consider the highest resolution in Tab. 8.1 and four coupling iterations. After four coupling iterations, the error is already reduced to $\varepsilon \approx 7 \cdot 10^{-5}$. In the sequential case, the resulting runtime was 26 minutes. With four processors, this time was reduced to 12 minutes[8] resulting in a 54% parallel efficiency. We also tested the iterative AztecOO-GMRES[9] solver in combination with the Ifpack-ILU [77] preconditioner. In the sequential case, they performed well, but in the parallel case, due to the actual configurations[10] of AztecOO, the efficiency was lower as with SuperLU-DIST. Therefore, future work will include finding and developing of an efficient preconditioner and iterative solver for Nitsche's method, which scale well in parallel and can tackle larger FSI problems on distributed memory systems.

8.3.2. 2D Transient Results

In the following, we consider two benchmark scenarios to verify our approach also for transient cases. Our approach to handle moving boundaries for Nitsche's method and the

[6] http://trilinos.sandia.gov/packages/amesos/
[7] http://trilinos.sandia.gov/packages/nox/
[8] On a quad-code Intel i7 2.9GHz machine
[9] http://trilinos.sandia.gov/packages/aztecoo/
[10] Improving these configurations is subject of future work.

implicit coupling methods has been presented above. These methods will be employed in the following to compute the FSI2 and FSI3 transient benchmarks [46].

FSI3

We start with the FSI3 scenario that is computable only with implicit coupling. The characteristic parameters for the FSI3 are set as $\bar{v}_f = 2\frac{m}{s}$, $\rho_s = 10^3\frac{kg}{m^3}$, $Re = 200$, and $E = 1.4 \cdot 10^6\frac{kg}{ms^2}$. The main reason for the necessity of implicit coupling is that both, the structure and the fluid, have the same density $\rho^f = \rho_s = 10^3 kg/m^3$. This scenario is described in [46] and above. It is the most commonly used transient benchmark, since only this transient scenario is considered in the review article [88] for various coupling approaches. The discrete time step is set to $\Delta t = 10^{-3}$ and the end simulation time is usually $6-7$ seconds, which results in $6000-7000$ time steps. Since we are using Aitken under-relaxation, per time step in average 11 coupling iterations were required.[11] Therefore, in total, the fluid and structure solvers were called $60000-70000$ times. This implies with a fluid and structure solving time of total 10 seconds a total simulation time of more than a week. For this reason, such transient FSI scenarios are computationally expensive. Therefore, we compute these scenarios with significantly less DoFs than the stationary FSI1. We show two resulting snapshots at times 5.6 and 5.65 seconds

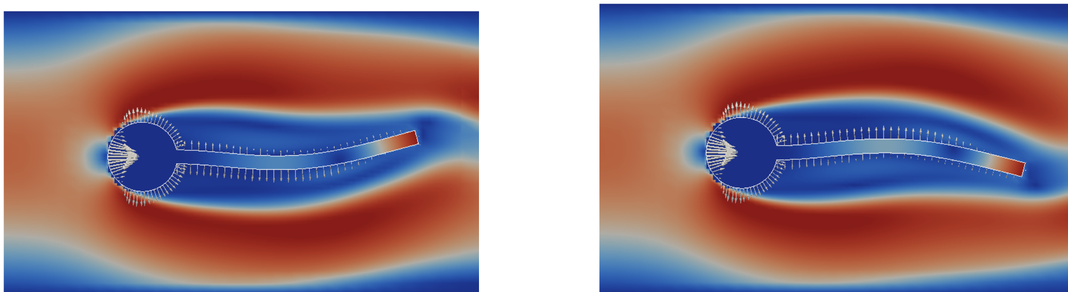

Figure 8.10.: Two different snapshots of the FSI3 simulation. The arrows on the white polygon represent the coupled stress vectors.

in Fig. 8.10. The structure is mapped into the flow field in Fig. 8.10, where the coloring of the structure represents its total displacement. The white line is the polygon with 136 points, and the white arrows on the polygon are the stress vectors $\underline{\sigma}_f \mathbf{n}$. In the first couple of seconds, the bar is almost stationary in the flow field, and then it starts to oscillate with increasing amplitude. The increasing amplitude saturates at around $3-4$ seconds. We measure the benchmark values for this case, which are the displacements at the tip of the bar and the total lift and drag forces. Since the scenario is transient, these values are time dependent. This is illustrated by the total drag and lift forces in Fig. 8.11. The forces were calculated by curve integrals. Therefore, and because

[11] Quasi-Newton methods would require less iterations.

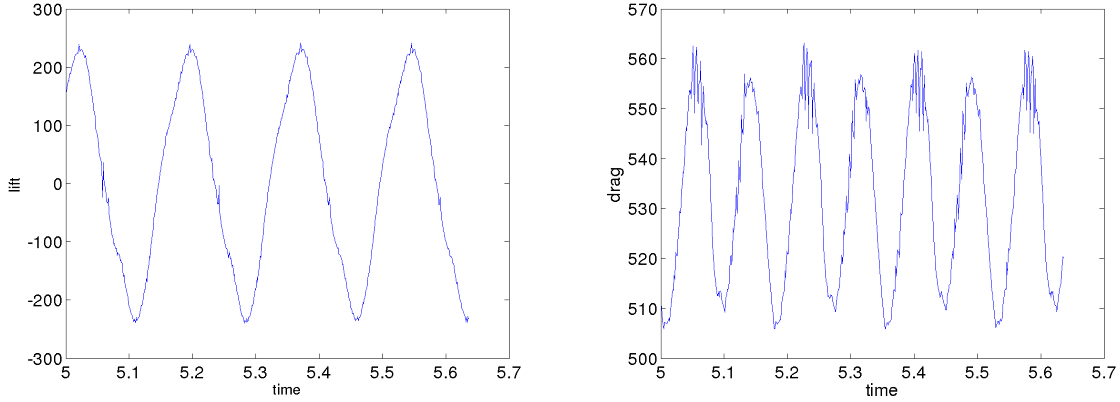

Figure 8.11.: Plots of the time variant total drag (right) and lift (left) forces for FSI3.

of the moving boundaries, they show a small high frequent noise. For each benchmark value, we measure the offset, the amplitude, and the frequency. The measured values are shown in Tab. 8.2 for two different resolutions. The main benchmark value is the vertical displacement Ay. For the higher resolution, this matches almost exactly the benchmark displacement. Only the frequency differs in this case by 8%. The displacements in x-direction also match the benchmark values well. In the case of the drag value, the offset is higher compared to the benchmark offset, but the amplitude matches with the benchmark lift amplitude. For the lift values, we notice that the resulting amplitude is 30% higher than the reference value. Overall, the results of FSI3 verify our approach for the moving boundaries and for transient FSI scenarios.

#Fl×#Sr×#Poly	Ax	Ay	Drag	Lift
2507 × 864 × 136	-0.02866 ±0.02857[11.1]	-0.00111 ±0.03301[5.5]	524.5 ±23.5[11.1]	56.50 ±214.50[5.5]
5133 × 864 × 136	-0.00288 ±0.00282[11.4]	0.00166 ±0.03452[5.7]	532 ±25[11.4]	-1.5 ±229.50[5.7]
Benchmark [46]	-0.00269 ±0.00253[10.9]	0.00148 ±0.03438[5.3]	457.3 ±22.66[10.9]	2.2 ±149.78[5.3]

Table 8.2.: Results of the FSI3 scenario. The offset, the amplitude, and the frequency are measured for each of the four quantities.

FSI2

Finally, we consider the FSI2 transient scenario. The characteristic parameters for the FSI2 are set as $\bar{v}_f = 1\frac{m}{s}$, $\rho_s = 10^4 \frac{kg}{m^3}$, $Re = 100$, and $E = 5.6 \cdot 10^6 \frac{kg}{ms^2}$. Even though the structure has ten times higher density than the fluid, the scenario requires implicit coupling. In addition, this scenario requires considerably longer simulation times, since the

#Fl×#Sr×#Poly	Ax	Ay	Drag	Lift
1964 × 864 × 136	-0.022064 ±0.022836[4.0]	0.001450 ±0.090450[2.0]	243.50 ±139.5[4.0]	9 ±382[2.0]
2091 × 864 × 136	-0.02207 ±0.0205[4.0]	0.00103 ±0.08937[2.0]	234 ±127[4.0]	7 ±376[2.0]
Benchmark [46]	-0.01458 ±0.01244[3.8]	0.00123 ±0.0803[2.0]	208 ±73.75[3.8]	0.88 ±234.2[2.0]

Table 8.3.: FSI2 results. The offset, the amplitude, and the frequency are measured for each of the four quantities.

bar starts oscillating only after more than 5 seconds. Therefore, we were able to compute this scenario only with considerably lower resolution than FSI3 as shown in Tab. 8.3. For this reason, the results in Tab. 8.3 differ more from the benchmark results, than in the previous case. The oscillating bar has an oscillation with lower frequency than in FSI3 but with higher amplitude. This is illustrated by the two snapshots in Fig. 8.12. Even with this lower resolution the error in the amplitude of Ay is around 10%. The frequency of the oscillation matches exactly the benchmark value.

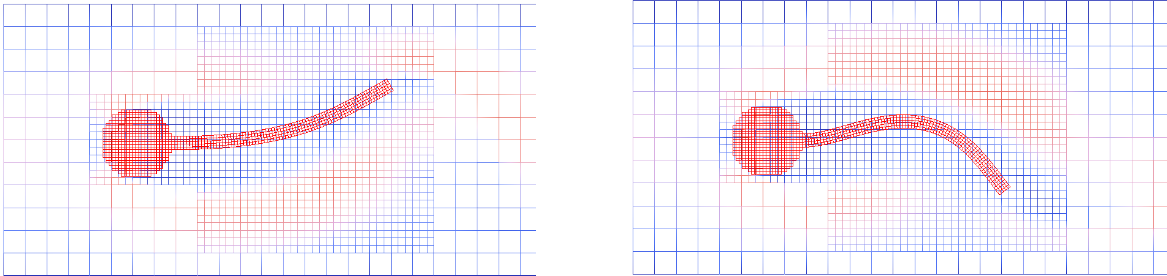

Figure 8.12.: Two different snapshots of the FSI2 simulation, with larger displacements as for FSI3. The illustrations show the two Cartesian meshes: the structure's mesh is red, and the fluid's mesh is colored by the magnitude of the velocity vector.

8.4. 3D Results

In the final section of this chapter, we consider two 3D FSI scenarios. In 3D, there are no standard benchmark scenarios available. Therefore, we set up a stationary and a transient 3D FSI scenario in order to verify the 3D cut-cell and boundary integral methods for various configurations. The first scenario is a stationary coupling, where a vertical bar is placed in a 3D flow channel. In the second scenario, a stationary sphere is placed

in a similar channel flow. This sphere is considered rigid and is accelerated by the flow. These two scenarios demonstrate the true potential of Nitsche's method and our cut-cell and boundary integral methods for 3D FSI scenarios, where a classical approach with unstructured meshes becomes even more costly than in 2D. For parallel computations in 3D, our approach implemented within Sundance has the same capability as in 2D, to automatically decompose both the structure and fluid mesh in a load balanced way and to solve both the structure and the fluid problem in parallel. This parallel capability has already been illustrated for the static FSI1 scenario, and will not be discussed here further (see Chapter 9 for 3D parallel simulations).

8.4.1. 3D Stationary Coupling Results

We consider the flow channel in Fig. 8.13 with a size of $2.5 \times 0.41 \times 0.41$. The fluid in this channel has a density of $10^3 \frac{kg}{m^3}$, similar to the 2D scenarios, and its parabolic inflow velocity is $0.45 \frac{m}{s}$ with $Re = 20$ as it is defined for 3D in [80] and as it was already used in Chapter 7. All other fluid parameters correspond to the static FSI1 scenario [46].

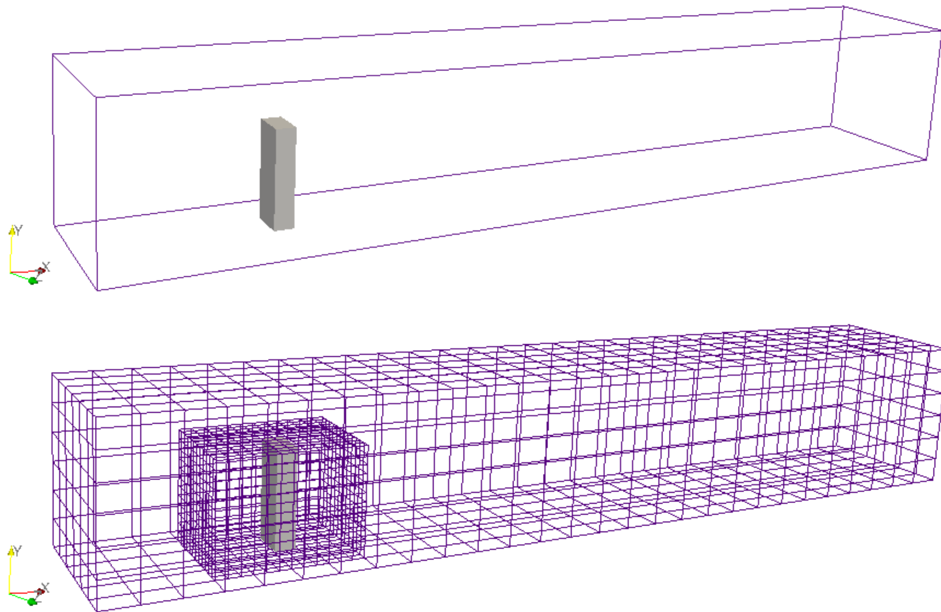

Figure 8.13.: Configuration of the static 3D FSI scenario (top). The mesh resolution in 3D was chosen as $27 \times 7 \times 7$ (bottom) with further refinement of the cells around the obstacle.

In this channel, we place a vertical elastic bar characterized by the Young modulus of $E = 0.4 \cdot 10^6 \frac{kg}{ms^2}$ and the Poisson ratio of $\nu_s = 0.4$. This bar has a size of $0.05 \times 0.25 \times 0.1$ and is placed at the position of $(0.45, 0.0, 0.125)$ into the flow channel. We use an initial 3D mesh resolution for the fluid, and further refine the cells in the rectangle area around

the vertical bar as shown in Fig. 8.13. For a spatial discretization similar to that in Chapter 7, we use stabilized Q_1Q_1 elements, which for the 3D-1Z scenario [80] worked well.

We apply the same stationary coupling algorithm with an under-relaxation factor of $w = 0.31$. The coupling surface is given by a triangulation with 620 triangles. Similar to the 2D case, we create one additional twin triangle surface to map the values between the Eulerian and Lagrangian frameworks. The stopping criterion for the coupling scheme was the tolerance of $\varepsilon = 10^{-6}$ in the displacements. The resulting displacements and flow fields after 17 coupling iterations are shown in Fig. 8.14. At the Lagrangian position of $(0.475, 0.25, 0.175)$ that represents the midpoint of the top face of the bar, we measure a displacement vector of $(1.1667e-2, -4.2293e-4, 6.619e-3)$. As expected, the displacement is in x-direction, whereas in z-direction, there is also a significant displacement due to the non-central position of the elastic bar. The resulting displaced structure and the coupling stress vectors with the triangular surfaces are shown in Fig. 8.14. Since this scenario does not have benchmarked values, we can conclude only based on the presented result that Nitsche's method with the developed 3D cut-cell and surface integral methods is not just converging but is also delivering reasonable results for this static 3D FSI scenario.

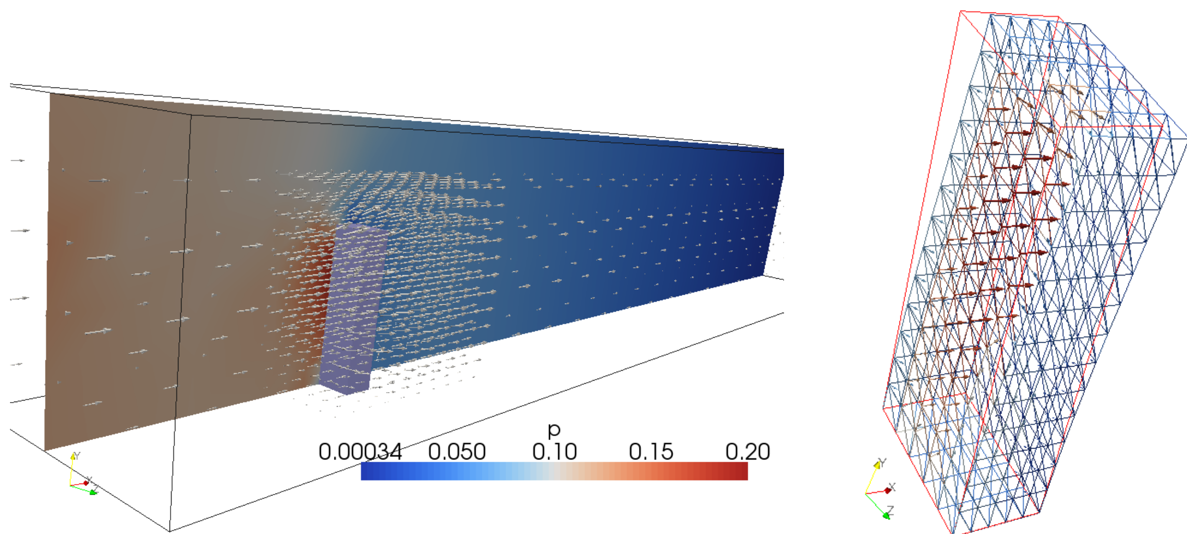

Figure 8.14.: Results of the static FSI scenario (left). The deformed structure including the triangulated surface with the coupling stress vectors is shown on the right.

8.4.2. 3D Explicit Coupling Results

As a final example, we consider a simple 3D transient FSI scenario. The structure is a sphere with dimensionless radius of one and is modeled as a point-wise mass of $1kg$. Therefore, no Sundance solver is required for the structure. The structure is placed in a channel flow with a size of $8 \times 4 \times 4$ and is fixed at the beginning. The fluid has a lower density compared to the previous case of only $1kg/m^3$, and the parabolic inflow velocity was set to $2.25m/s$. In the first 0.2 seconds of the simulation, the sphere is fixed, but after this it is freely accelerated by the forces in the x-, y-, and z-directions. These forces are computed by surface integrals of the respective stress vector components on the sphere's surface. The fluid's mesh is a regular Cartesian mesh with a $13 \times 9 \times 9$ resolution as illustrated in Fig. 8.16. The boundary surface is shown in Fig. 8.15. It is formed by 500 triangles and 252 nodes. This figure also shows the coupling stress vectors for a given snapshot, which push the sphere in the x-direction along the flow field.

Due to the simple structure modeling, higher structure density, and the relatively small sphere radius, this scenario allowed for an explicit coupling by the presented staggered scheme.

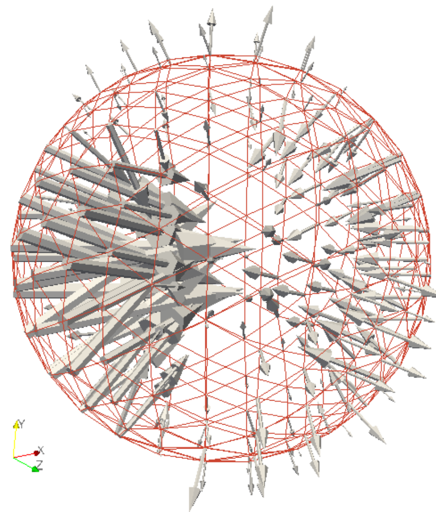

Figure 8.15.: Illustration of the triangular surface with 500 cells representing the sphere. The vectors on the nodes represent the corresponding stress vectors computed from the flow field.

Two snapshots in Fig. 8.16 show the simulation results, where the sphere moves from the left to the right. During this movement, the sphere passes several fluid cells and the triangular surface might intersect some of the cells in an irregular way.[12]

The sphere, however, is moving along the flow direction smoothly, which shows that the

[12] One irregular case is when the surface intersects one edge of a brick cell more than once. See Chapter 7 for more detail.

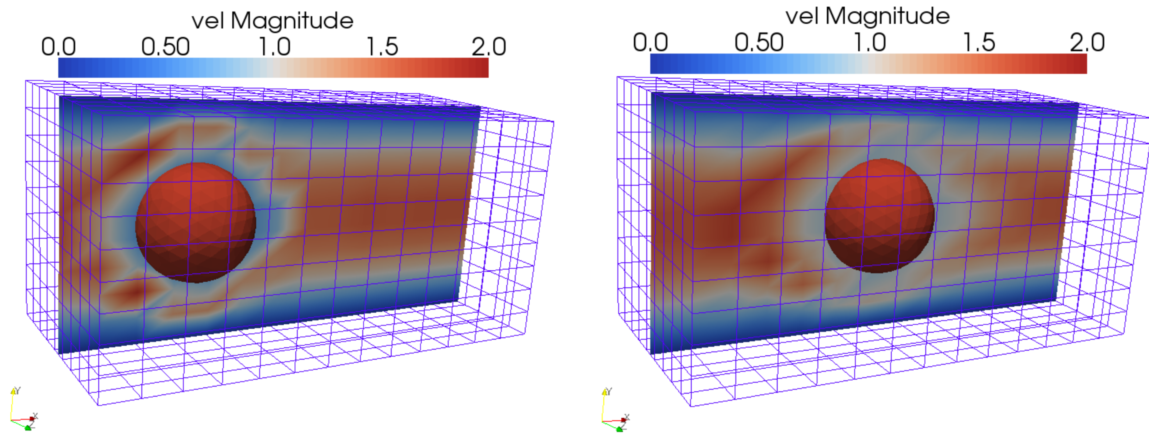

Figure 8.16.: Two snapshots of the explicit coupling. The sphere is accelerated and moved along the flow field.

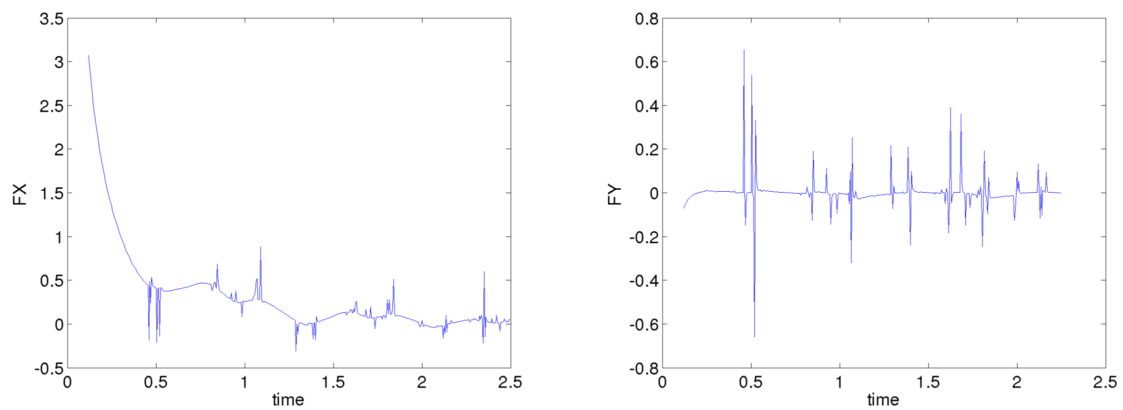

Figure 8.17.: Total forces in x-direction (left) and total measured forces in y-direction (right).

irregular intersections for the cut-cell and the surface integral in 3D are handled well. The homogeneous velocity (v_X, v_Y, v_Z) of the sphere is changed by the measured total forces (F_x, F_y, F_z) on the surface such that

$$v_X = v_X + \Delta t \frac{F_x}{m}$$

$$v_Y = v_Y + \Delta t \frac{F_y}{m}$$

$$v_Z = v_Z + \Delta t \frac{F_z}{m},$$

where m is the mass of the sphere that was set to $1kg$. The time step was set to $\Delta t = 0.005$. We plot the forces that vary in time in Fig. 8.17. The x-component of the force vector has a higher initial value, since the sphere is fixed. As the obstacle

is accelerated, F_x is reduced to zero whereas F_y is always approximately zero. Several peaks in the forces are visible in Fig. 8.17, when the boundary crosses cells. However, these peaks are limited in size and time and therefore, do not influence the results much.

9. Porous Media Simulation with the Stokes-Brinkman Model

In order to demonstrate the generality and the usability of the implemented IB features within Sundance, we investigate another type of IB method in this chapter. This method was already introduced in Chapter 4, where it was called the volume penalty method. For viscous flows, the method is called Stokes-Brinkman or Navier-Stokes-Brinkman method. We employ it for the simulation of porous media, where our goal is to determine the permeability of a given medium. For this type of flow simulation, we are not interested in the boundary values of the flow, but only in the overall flow field. Therefore, an inconsistent method such as the chosen volume penalty method proves to be suitable and more efficient then the previously used Nitsche's method. In 3D, we model the porous medium with a package of spheres that touch each other at given points, where each sphere represents a sand grain, whereas in 2D, we model the porous medium by a package of circles, which do not touch each other. We verify the volume penalty approach by imposing the BC in the classical way. Then, we compare the resulting flow rates to the Stokes-Brinkman approach.

In the final section of this chapter, we test the parallel scalability of the 3D adaptive Cartesian mesh, DoF map, the pre-fill transformation, and other components, which we developed in this thesis for the Sundance toolbox. We show strong scaling results for the Stokes-Brinkman simulation in 3D with up to 192 processors. The Stokes-Brinkman approach and the strong scaling results are presented also in our recent publication [18].

9.1. The Governing Equation and the Geometry Model

For all the computations we consider only the Stokes-Brinkman equation to simulate the porous medium on the micro scale[1], where the goal is to determine the permeability of the medium. The model has already been presented in Chapter 4 and was used in [75, 2] in a similar context. We model the porous medium on this micro scale as a channel filled with sand grains that have a diameter of less than a millimeter. On these scales, the resulting Reynolds number is of order $10^{-3} - 10^{-5}$ for water. Hence, the convection term in the case of Navier-Stokes equations could be neglected. In order to model the

[1] The micro scale in our case is the size of several millimeters.

boundaries, we use the Stokes-Brinkman equation

$$-\mu\Delta\mathbf{u} + \nabla p - \mathbf{k}(\mathbf{x})^{-\mathbf{1}}\mu\mathbf{u} = 0, \qquad in\ \Omega_f \tag{9.1}$$
$$\nabla \cdot \mathbf{u} = 0, \qquad in\ \Omega_f, \tag{9.2}$$

where μ is the fluid's viscosity and is indirectly proportional to the Reynolds number $\mu \approx \dfrac{1}{Re}$. For all the computations, we choose $Re = 10^{-3}$. As defined in Chapter 4, $\mathbf{k}(\mathbf{x})$ represents the local permeability of the medium that varies locally in order to approximately impose the zero Dirichlet BC on the surface of the sand grains by assigning these grains with a very low permeability:

$$\mathbf{k}(\mathbf{x}) = \begin{cases} (k_F, k_F) & \mathbf{x} \in \Omega_f \\ (k_S, k_S) & \text{else } (\mathbf{x} \in \Omega_s). \end{cases} \tag{9.3}$$

The last term in the impulse equation (9.1) with the defined permeability coefficient acts as a penalty or *slowdown* term that forces the velocity at a position $\mathbf{x}$ with low $\mathbf{k}(\mathbf{x})$ to nearly zero. In the structure domain Ω_s, represented by the sand grains, the permeability is set to $k_S = 10^{-5}$, such that the flow is almost completely stopped, whereas in Ω_f $k_F \longrightarrow \infty$ that transforms (9.1) to the Stokes equations for the fluid region. In order to ensure convergence of this method in the implementation, we multiply the penalty term $\frac{1}{\mathbf{k}}$ with the factor $\frac{1}{h}$, where h is the diameter of the cell. This way, when $h \to 0$, then $\frac{1}{h\mathbf{k}} \to \infty$ ensuring convergence for higher resolutions.

The sand grains are modeled by analytically described standard geometries, since our goal is not to model individual shapes of the sand grains. In addition, the analytically described geometries are more efficient than the polygon in 2D or the triangular surface in 3D. Therefore, we model in 2D the sand grains by circles that can not touch each other, in order to form a free channel between them, where the fluid can flow. In 3D, we model the sand grains by spheres with a given radius, but here the spheres form a compact package. We created a class in Sundance that can contain several geometries called *CurveCollection*. This is illustrated in Code 22. The scenarios that we compute will be illustrated in the next section for 2D and 3D.

Trapezoidal integral

In Chapter 7, we developed the cut-cell and boundary integral methods that work well when the cell is intersected in a regular way. In 3D, if we consider the compact sphere package, where the spheres touch each other, these methods can not be employed with coarser mesh resolution. Moreover, these methods are computational costly. Since we are interested only in the overall flow field of this scenario, a cheaper integral method could be employed here. Further, we notice that the penalty term in (9.1) does not have to be computed up to machine precision, as this was the case for Nitsche's method. Therefore, we use the trapezoidal quadrature rule for the penalty term's integration, where per

dimension p points are evaluated. In 2D, this implies p^2 and in 3D p^3 quadrature points. The weight of each quadrature point depends on its coordinates:

$$\omega(\mathbf{x}) = \begin{cases} \alpha_1 \omega_T & \mathbf{x} \in \Omega_f \\ \alpha_2 \omega_T & \text{else } (\mathbf{x} \in \Omega_s), \end{cases} \tag{9.4}$$

where ω_T is the standard trapezoidal weight, α_1 is the weight factor of the fluid that is set to $\alpha_1 = 10^{-8}$, and α_2 is the weight factor of the structure domain that we set to $\alpha_2 = 1$. In Ω_f, the factor $\alpha_1 = 10^{-8}$ ensures that the volume penalty term almost vanishes from (9.1), by setting the permeability to a high value. On the other side the weight $\alpha_2 = 1$ in Ω_s enables the penalization of the flow field. Similar to the cut-cell method this results in a set of special weights for each cell that is intersected, and these weights are handled similarly to the ones generated by the cut-cell method. The Sundance code for this type of quadrature is simple and is shown in Code 22.

Code 22 Sundance code to show the usage of *CurveCollection* and *TrapezoidQuadrature* classes.

```
Array<ParametrizedCurve> curves(0);
CurveBase *tmp;
CurveCollection *curveCollect = new CurveCollection( 1e-8 , 1e-0 , 1);
ParametrizedCurve curve = curveCollect;
curveCollect->addCurve(new Circle(0.2,0.167,0.13,outV,inV ));
curveCollect->addCurve(new Circle(0.2,0.5  ,0.13,outV,inV ));
...
QuadratureFamily tquad = new TrapezoidQuadrature(8);
Expr h=new CellDiameterExpr();
Expr ka=Integral(OnCircle, k*(1/Re)*(1/h)*(ux*vx + uy*vy),tquad,curve);
```

9.2. Computational Results in 2D and 3D

In this section, we show the computational results for a 2D and a 3D example, where the main goal is to verify the approach to model the fluid with the Stokes-Brinkman equation. The quantity of interest is the total flow rate through the channel that can be further used to compute the average permeability of the channel. In order to have reference values, we compute each scenario with two standard approaches imposing the zero Dirichlet BC on the facets of the Cartesian mesh. The first approach considers the intersected cells as pure fluid cells, and only those cells are structure, which are completely inside the structure domain. This approach, we denote by *Stokes H*, since this approach gives an upper limit of the flow rate. The opposite of this approach is to consider all intersected cells as only structure cells. Accordingly, this approach is denoted by *Stokes L* and gives a lower limit of the flow rate. We expect that the resulting flow

rate of the Stokes-Brinkman model is between these two limits and even for lower mesh resolution the flow rate is accurately computed. It is important to underline here that for near boundary phenomenons such as drag and lift forces computations, the Stokes-Brinkman model would give inconsistent results. For such cases, a consistent method should be employed. Here, however, the overall flow field is of interest, therefore, we use the presented approach. In comparison, this approach is computationally much cheaper than Nitsche's method that, e.g., for the sphere package needs a considerably higher mesh resolution to ensure that most of the intersected cells fall into the regular category (see Chapter 7).

9.2.1. 2D Results

In 2D, we use a simple representation of the geometry that is formed by 10 circles as illustrated in Fig. 9.1. The flow channel is the unit square $[0,1]^2$, where the 10 circles represent the obstacles. The flow is driven by a pressure Neumann BC[2] having the value 2 that is imposed on the left side of the channel. At the top and bottom walls, we impose zero Dirichlet BCs in the classical way, and the outflow is measured on the right side of the channel. We used Q_2Q_1 elements that do not require stabilization.

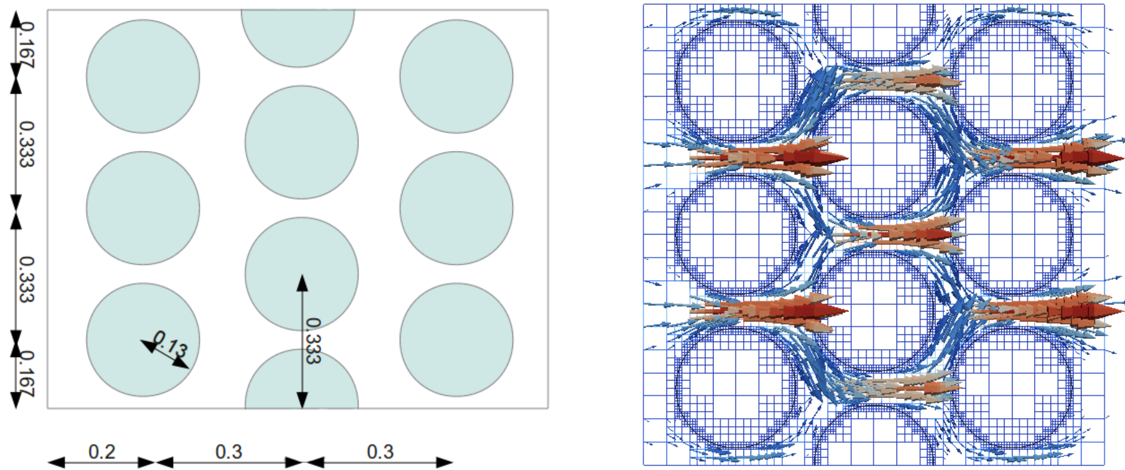

Figure 9.1.: Illustration of the scenario in 2D (left) and the resulting flow field (right) with a refined mesh. The circles (right) are colored with black within the flow channel of $[0,1]^2$.

Here, the main purpose is not to model the porous medium accurately, but to verify the Stokes-Brinkman approach. The resulting flow rates for various resolutions are shown in Tab. 9.1. We notice that the lower bound of the flow rate, computed by the *Stokes L* approach, is constantly increasing with increasing resolution, whereas the upper bound *Stokes H* is continuously decreasing with higher mesh resolution. These limits specify

[2]The derivative of the pressure in the normal direction has the given value.

#Cells, Resolution	Stokes L	Stokes-Brinkman	Stokes H
1840, 20×20, $l = 1$	1.96e-07	3.48e-07	6.84e-07
6244, 50×50, $l = 1$	2.46e-07	3.75e-07	4.55e-07
6640, 20×20, $l = 2$	2.59e-07	3.76e-07	4.38e-07
17336, 100×100, $l = 1$	2.85e-07	3.72e-07	4.15e-07
19012, 50×50, $l = 2$	3.32e-07	3.79e-07	4.03e-07
43632, 100×100, $l = 2$	3.42e-07	3.73e-07	3.85e-07
107096, 200×200, $l = 2$	3.53e-07	3.68e-07	3.73e-07

Table 9.1.: The flow rates measured for the 2D scenario. l represents the refinement level of the mesh at the boundary.

the interval where the *Stokes-Brinkman* flow rate should be included, and this is satisfied for all resolutions. Further, we notice that Stokes-Brinkman approximates the flow rate well even for lower mesh resolutions, where the difference between the lower and upper bound of the flow rate is relative high. These results verify our approach to handle IBs that we apply in the following for 3D.

9.2.2. 3D Results

In 3D, we consider a flow channel of size $0.96 \times 0.94 \times 0.83$. On the left side of the channel, a pressure Neumann BC with the value 2 drives the flow, where on the right side of the cube the outflow rate is measured. All other four walls have zero Dirichlet BCs. Here, we model the porous medium by 40 spheres[3] of radius 0.12. These spheres form a compact package, where the spheres touch each other such that no further compression is possible. This configuration of the spheres and the resulting flow field are illustrated in Fig. 9.2.

The Stokes-Brinkman equations are solved in the similar way in 3D as in 2D, but in contrast to the 2D case, we use Q_1Q_1 elements that require stabilization. We use the PSPG stabilization that was presented in Chapter 3. This stabilization results in a simple pressure stabilization term for the stationary Stokes equations with Q_1Q_1 elements. We computed the flow field and the resulting flow rate for different mesh resolutions. Analog to 2D, we determined the lower and upper bound of the flow rate with the *Stokes L* and the *Stokes H* approach, respectively. The results are presented in Tab. 9.2 for three different mesh resolutions.

For the highest resolution, the difference between the lower and upper bound of the flow rate is significant, but the measured Stokes-Brinkman flow rate is always between these bounds. The change of the Stokes-Brinkman flow rate in the last refinement step is also minor in comparison to the other two approaches, therefore, we can conclude that this

[3]Special thanks to Lieb Michael M.Sc. for the configuration.

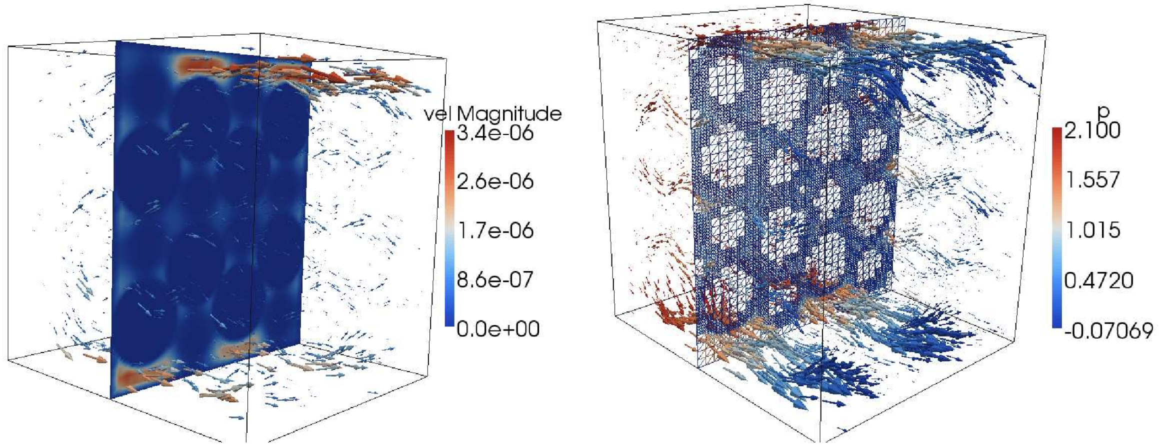

Figure 9.2.: The flow channel $0.96 \times 0.94 \times 0.83$ with the resulting flow field (left). The mesh with an initial resolution of $18 \times 18 \times 13$ in the illustration (right) is refined at the boundary using one further level.

#Cells, Resolution	Stokes L	Stokes-Brinkman	Stokes H
4212, $18 \times 18 \times 13,\ l = 0$	9.93e-09	2.59e-07	8.62e-07
79144, $18 \times 18 \times 13,\ l = 1$	3.50e-08	2.09e-07	2.93e-07
433126, $36 \times 36 \times 28,\ l = 1$	7.55e-08	2.07e-07	2.26e-07

Table 9.2.: The flow rates measured for the 3D scenario. l represents the refinement level of the mesh at the boundary.

approach approximates the overall flow field in 3D more efficiently than the classical BC methods on Cartesian meshes, without using consistent IB methods.

9.3. Strong Scaling Results of the 3D Parallel Computations

In the last section of this chapter, we use the presented 3D scenario to test the adaptive Cartesian mesh, the associated DoF map, and the pre-fill transformation for parallel simulations. Due to the actual implementational limitations of our Cartesian meshes we are limited to the a given maximal problem size, since the mesh is globally present on each processor (see Chapter 6 for more details). However, we underline here that the interfaces and the developed concept allow massively parallel simulations. The implementation or the integration of Cartesian meshes that do not have global storage, such as p4est[27] or Peano[92], would enable Sundance to compute larger problems than the ones we compute here.

All the computations were performed on the MPP cluster of the Leibniz Supercomputing Center (LRZ) in Garching[4] on Optoron 2.6GHz AMD processors. The architecture of this cluster is a fat-tree that causes a communication bottleneck beyond 64 processors.[5] However, in this section, we still achieved good speedup and efficiency beyond 64 processors. In the following, we compute two 3D problems. The first one with Q_1Q_1 stabilized elements from the previous section, and the second example is the same scenario but with Q_2Q_1 elements and with fewer cells.

9.3.1. Results with Q_1Q_1 Elements

We use the presented example from the previous section in Fig. 9.2 with the highest resolution. With an initial resolution of $36 \times 36 \times 28$ and a one level refinement on the boundary surface, the resulting mesh has 433,126 cells, which with the Q_1Q_1 element results in approximately $2.5 \cdot 10^6$ unknowns. This system is solved with the TSF-BiCGStab solver and the standard ILU preconditioner from the Trilinos library [43] that Sundance is also part of. These packages are rather simple solvers but they have good parallel scalability. Other, more efficient, parallel solvers are available for Sundance within the Trilinos library, such as the Aztec-GMRES with ML-AMG preconditioner [34] or even external solvers such as the SuperLU-Dist [57]. Since we are mainly testing here the mesh, DoF map, and other Sundance components' parallel capabilities, we do not focus on the parallel solver. The strong scaling results for the chosen solver and for Q_1Q_1 elements are presented in Tab. 9.3.

Nr. Proc.	1	2	4	8	16	32	64	128
assembly time (sec.)	682	356.4	183.3	98.1	51.6	27.84	17.66	10.3
solver time (sec.)	875	497.3	238.4	135.5	91	49.11	36.2	20.97
total time (sec.)	1620	893	448	254	161	94	70	48

Table 9.3.: Parallel execution time of the Stokes-Brinkman problem with Q_1Q_1 elements

Besides the total runtime of the simulation, we show the average assembly and solver time in Tab. 9.3. The average assembly time is the main indicator how well the implemented DoF map and the pre-fill transformation work for parallel matrix assembly, whereas the total runtime mainly reflects the load-balanced partitioning of the problem among the processors. A concrete mesh partition is shown in Fig. 9.3 for a parallel run with 32 processors. Due to the coarse cell partitioning of the mesh and the various numbers of ghost cells on processors that do not count as load, the individual assembly and solver times vary among the processors. The total time that we measure includes besides the matrix assembly and solving additional operations (e.g., mesh set-up and refinement).

[4] http://www.lrz.de/services/compute/linux-cluster/overview/

[5] The interconnection between the sub trees of 64 processors has a low bandwidth.

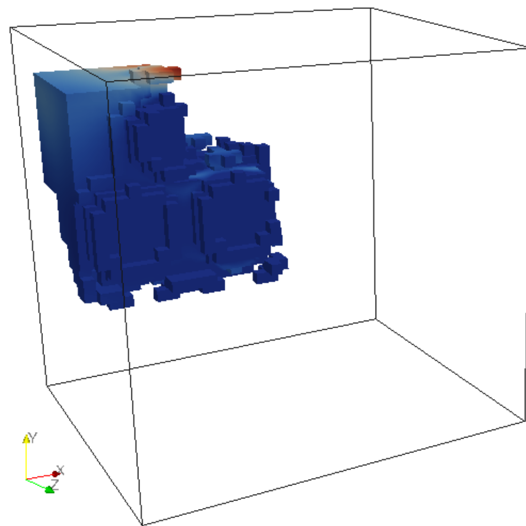

Figure 9.3.: Mesh decomposition with 32 processors. The figure shows the mesh belonging to the 10th processor.

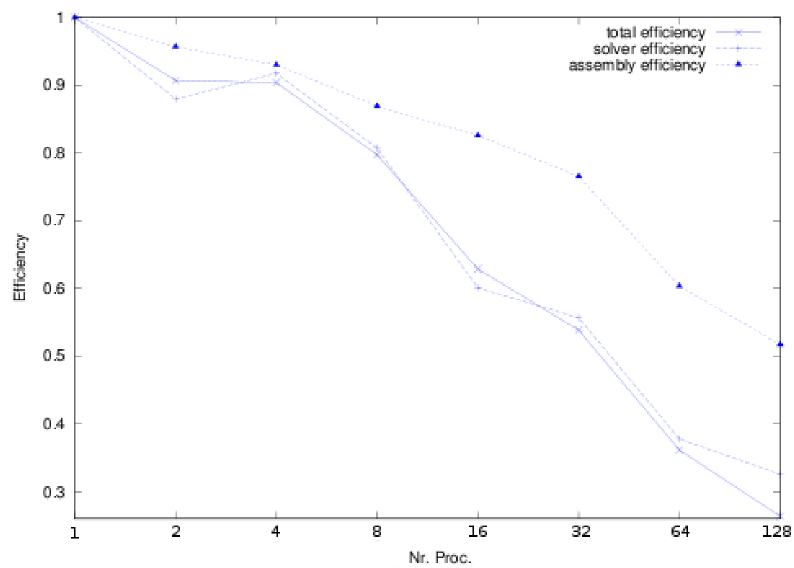

Figure 9.4.: Efficiency of the parallel Stokes-Brinkman simulation with Q_1Q_1 elements.

Once the sum of the average assembly and solver time is considerably less than the measured total runtime, this is an indication for bad load-balancing or for a bottleneck in the computation. The resulting efficiency for up to 128 processors is shown in Fig. 9.4. By default, the sequential run is used as a reference with 100% efficiency. With up to 8 processors, we observe only a slight decrease in the efficiency, where the total efficiency with 8 processors is still 80%. With 128 processors the overall efficiency decreases to 28% that is mostly due to the solver and the cluster's architecture. Even in this case, as Tab. 9.3 shows, we can further reduce the total runtime in comparison to 64 processors. In Fig. 9.4, we also notice that the assembly efficiency is always higher than the solver's efficiency. Even even with 128 processors, it is still above 50%, where in average one processor owns $10,000$ cells. The high efficiency of the matrix assembly shows the parallel scalability of our Cartesian mesh, DoF map, and pre-fill transformation implementation.

9.3.2. Results with Q_2Q_1 Elements

We further test our Sundance implementation for the higher order element Q_2Q_1, where the three velocity components are represented by a quadratic basis in 3D. Therefore, the number of local DoFs in an element increases significantly, from 32 with Q_1Q_1 to 89 with Q_2Q_1. This results also in a larger element matrix and, accordingly, in a global matrix with considerably larger bandwidth. For these reasons, we consider the previous scenario with the same solvers and setting but with only $250,047$ elements that results in a system with approximately $1 \cdot 10^7$ global DoFs. Even though we have approximately only four times more unknowns than in the Q_1Q_1 case, due to the denser system matrix, the memory required increases by more than ten times.

Nr. Proc.	1	2	4	8	16	32	64	128	192
assembly time (sec.)	1542	586	330	178	101	58.7	35.2	26	21.6
solver time (sec.)	1930	1077	701	318	201	104.4	60.4	36.5	31.2
total time (sec.)	3539	1667	1036	498	304	168	100	68	59

Table 9.4.: Parallel execution time of the Stokes-Brinkman problem with $250,047$ Q_2Q_1 elements.

The resulting runtimes for the Stokes-Brinkman problem with Q_2Q_1 elements in 3D are shown in Tab. 9.4. Analog to the previous case, we show the average assembly and solving times, whereas the total time represents the measured computation time. The resulting parallel efficiency for these tests is shown in Fig. 9.5. In the first steps of the strong scaling study, we notice a superlinear speedup. In Chapter 6, we compared the three available mesh types in Sundance, and showed that the serial implementation of the Cartesian mesh gives the shortest runtimes for 2D and 3D Poisson problems. Therefore, the superlinear speedup in Fig. 9.5 is not caused by a poor sequential implementation, but this is mostly due to cache effects caused by the larger bandwidth of the global

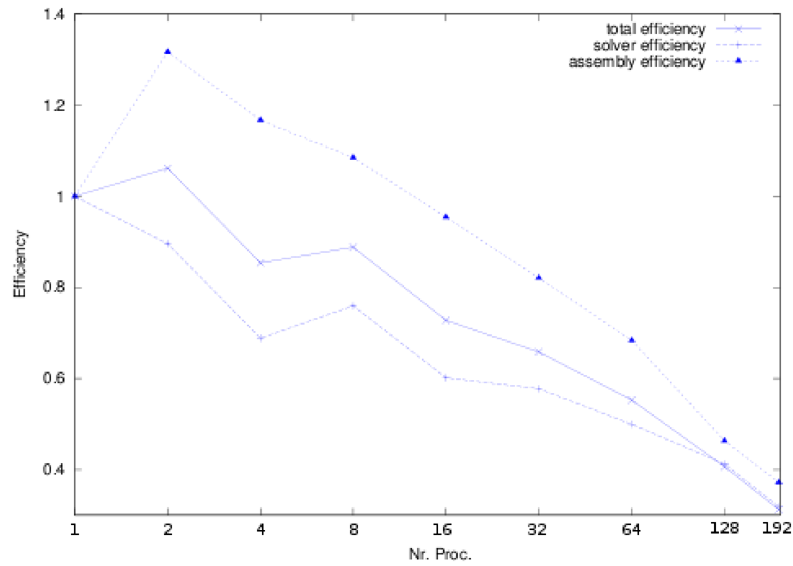

Figure 9.5.: Efficiency of the parallel Stokes-Brinkman computations with $250,047$ Q_2Q_1 elements.

matrix. This effect is only present in the assembly time, where the measured assembly efficiency with 8 processors is above 100%. With increasing processor numbers the total efficiency is decreasing, but even with 192 processors the assembly efficiency is still around 40%. We also notice that in the step from 64 to 128 processors the decrease in the efficiency is significant and it is mainly due to the MPP cluster's architecture. The decrease in the efficiency for higher processor numbers is also due to the higher number of ghost cells, which induce additional computations compared to the sequential case. Similar to the previous tests, the efficiency of the assembly processes is higher than the solver efficiency, especially for lower processor numbers.

These results show that the matrix assembly process even with the hanging facet handling is scaling well. In particular, the matrix assembly including the pre-fill element transformation shows good scaling. The efficiency of the parallel solver does not depend on Sundance, since they are separate packages. In order to increase the overall efficiency of such computations in Sundance, a problem tuned linear solver should be employed that has a better parallel scaling than the one used in these computations.

10. Summary and Outlook

Finally, we summarize the achieved results and the developed methods of this thesis, and give an outlook on future research and development directions.

10.1. Summary

In this thesis, we mainly focused on structured adaptive Cartesian meshes in combination with various IB methods and their implementations within a FEM-based PDE toolbox. For our implementation, we chose as baseline the Sundance PDE toolbox software that is part of the Trilinos library [43]. First, we extended this toolbox with rectangular elements and with parallel adaptive Cartesian meshes in 2D and 3D. For the parallel case, the Cartesian mesh is decomposed based on the Z-curve in a load-balanced way. To ensure continuity between elements of different refinement levels, we developed and implemented the so-called pre-fill transformation. It makes the necessary restriction on such elements, while at the same time preserves the architecture of a classical FEM toolbox. Furthermore, it does not create a bottleneck for parallel simulations on distributed memory systems. Second, we developed and integrated capabilities for IB methods that are formulated in a weak form and impose BCs weakly. The first component of these capabilities is an explicit representation of the boundary geometry, since the mesh's facets are not representing the boundary in this context. For this purpose, we deployed an analytical description of the geometry in 2D and 3D. Besides this, we implemented polygons in 2D and triangular surfaces in 3D for complex geometry representation. The requirements for weakly imposed BC on IB methods are the ability to compute volume and boundary integrals. We developed various cut-cell integrations for accurate volume integrations on mesh cells intersected by the boundary. Consistently to the cut-cell integrations, we developed boundary integration methods within Sundance.

With these developments, we were able to test and compute various IB methods within the frame of Sundance. Such a method is Nitsche's method for the Navier-Stokes equations that we applied here for the first time in an IB context. We further developed a simple approach to apply this method for transient scenarios and for moving boundaries in the flow field, and applied this approach to perform FSI simulations. We verified our approach by computing benchmark drag and lift values of stationary and transient Navier-Stokes simulations for the 'flow around the cylinder' [80]. Furthermore, benchmark FSI simulations validated our approach to handle moving boundaries and to

accurately compute forces on the boundary. Hence, we proved in this thesis the applicability of Nitsche's method in an IB context for FSI applications in 2D and in 3D as well. In addition, we computed the flow rate in a porous medium modeled by the Stokes-Brinkman equation. Here, we showed the applicability of volume penalty methods to impose no-slip BCs of viscous flows, where the detailed near boundary phenomena are not of interest. For this application, we also demonstrated the parallel capability of the developed methods and meshes by a good strong scaling.

10.2. Outlook

Sundance, as a FEM-based toolbox, offers not only fast prototyping and testing of new methods formulated in a weak form, but is also capable of efficient parallel computations by using an efficient preconditioner and linear solver from the Trilinos package. Therefore, Sundance is suited for the simulation of different applications that could be computed in future work within the frame of this toolbox. For efficient and parallel computation within Sundance, one needs to choose a suitable iterative solver that needs to be specially tuned for a given problem. This task is left for future work for Nitsche's method for the Navier-Stokes equations, since most flow scenarios that we computed did not require parallel computation. On the other hand, the developed Nitsche approach could also be integrated in other HPC research software such as the Peano CFD solver [26] developed at our Chair. The capabilities of Sundance could be extended in future developments with features such as Discontinuous Galerkin discretization, multiple meshes within a problem, and unknown fields that are defined only on a subdomain of the mesh. We mentioned in this thesis that the actual implementation of Cartesian meshes represents a storage bottleneck in parallel computations. However, in future developments, we plan to eliminate this bottleneck, while we also plan to improve the load-balancing algorithm of the mesh by partitioning the mesh's cells at each level (similar to p4est[27]). With such improved meshes, one could compute in Sundance problems with more than a hundred million unknowns. For FSI simulations, Sundance allows for the implementation of various approaches, different from the one used in this thesis. Future research could be done on monolithic approaches, where the structure is transformed to the Eulerian framework, and both problems are solved on the same mesh, similar to the IP approach in [31]. Alternatively, in future research, one could linearize both equations in Sundance and, in a partitioned approach, couple the structure and the fluid in each linear step, such that the nonlinear solver acts as an outer iteration comprising the whole coupled system.

A. Appendix

A.1. Notations for Structural Mechanics

The relation between the traction vector $\mathbf{t}$ and the stress tensor $\underline{\sigma}_s$ in 2D is denoted as

$$\begin{pmatrix} \frac{\partial}{\partial x_1} & 0 & \frac{\partial}{\partial x_2} \\ 0 & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} \end{pmatrix} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \tag{A.1}$$

and in 3D case

$$\begin{pmatrix} \frac{\partial}{\partial x_1} & 0 & 0 & \frac{\partial}{\partial x_2} & 0 & \frac{\partial}{\partial x_3} \\ 0 & \frac{\partial}{\partial x_2} & 0 & \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_3} & 0 \\ 0 & 0 & \frac{\partial}{\partial x_3} & 0 & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} \end{pmatrix} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{13} \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{A.2}$$

The matrix form of C, representing the relation between stresses and strains (as a fourth order tensor), in 2D is

$$C = \frac{E}{(1+\nu_s)(1-2\nu_s)} \begin{pmatrix} 1-\nu_s & \nu_s & 0 & 0 \\ \nu_s & 1-\nu_s & 0 & 0 \\ 0 & 0 & 1-2\nu_s & 0 \\ 0 & 0 & 0 & 1-2\nu_s \end{pmatrix}, \tag{A.3}$$

and in 3D:

$$C = \frac{E}{(1+\nu_s)(1-2\nu_s)} \begin{pmatrix} 1-\nu_s & \nu_s & \nu_s & 0 & 0 & 0 \\ \nu_s & 1-\nu_s & \nu_s & 0 & 0 & 0 \\ \nu_s & \nu_s & 1-\nu_s & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu_s & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu_s & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu_s \end{pmatrix}, \tag{A.4}$$

where E is the Young modolus and ν_s is the Poisson ratio.

The strain-displacement relation that is also denoted with non-linear operator $L_n(\mathbf{u})$, in 2D has the form of

$$\underline{\varepsilon}_s = \frac{1}{2}\begin{pmatrix} u_{1,1} & u_{2,1} \\ u_{1,2} & u_{2,2} \end{pmatrix} + \frac{1}{2}\begin{pmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{pmatrix} + \frac{1}{2}\begin{pmatrix} u_{1,1}\cdot u_{1,1} + u_{2,1}u_{2,1} & u_{1,1}u_{2,1} + u_{1,2}u_{2,2} \\ u_{1,1}\cdot u_{2,1} + u_{1,2}u_{2,2} & u_{2,2}u_{2,2} + u_{1,2}u_{1,2} \end{pmatrix}$$

$$\underline{\varepsilon}_s = \frac{1}{2}\begin{pmatrix} 2u_{1,1} + u_{1,1}^2 + u_{2,1}^2 & u_{1,2} + u_{2,1} + u_{1,1}u_{1,2} + u_{2,1}u_{2,2} \\ u_{1,2} + u_{2,1} + u_{1,1}u_{1,2} + u_{2,1}u_{2,2} & 2u_{2,2} + u_{2,2}^2 + u_{1,2}^2 \end{pmatrix}.$$

Component wise, this result in the following relations

$$\varepsilon_{11} = u_{1,1} + \frac{1}{2}\left(u_{1,1}^2 + u_{2,1}^2\right)$$

$$\varepsilon_{22} = u_{2,2} + \frac{1}{2}\left(u_{2,2}^2 + u_{1,2}^2\right)$$

$$\varepsilon_{12} = \varepsilon_{21} = \frac{1}{2}\left(u_{1,2} + u_{2,1} + u_{1,1}u_{1,2} + u_{2,1}u_{2,2}\right).$$

In 3D, the same relation in matrix form is

$$\frac{1}{2}\underline{\varepsilon}_s = \begin{pmatrix} u_{1,1} & u_{2,1} & u_{3,1} \\ u_{1,2} & u_{2,2} & u_{3,2} \\ u_{1,3} & u_{2,3} & u_{3,3} \end{pmatrix} + \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{3,2} & u_{3,3} \end{pmatrix} + \begin{pmatrix} u_{1,1} & u_{2,1} & u_{3,1} \\ u_{1,2} & u_{2,2} & u_{3,2} \\ u_{1,3} & u_{2,3} & u_{3,3} \end{pmatrix}\begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{3,2} & u_{3,3} \end{pmatrix}$$

$$\frac{1}{2}\underline{\varepsilon}_s = \begin{pmatrix} 2u_{1,1} & u_{2,1} + u_{1,2} & u_{3,1} + u_{1,3} \\ u_{1,2} + u_{2,1} & 2u_{2,2} & u_{3,2} + u_{2,3} \\ u_{1,3} + u_{3,1} & u_{2,3} + u_{3,2} & 2u_{3,3} \end{pmatrix} +$$

$$\begin{pmatrix} u_{1,1}^2 + u_{2,1}^2 + u_{3,1}^2 & u_{1,1}u_{1,2} + u_{2,1}u_{2,2} + u_{3,1}u_{3,2} & u_{1,1}u_{1,3} + u_{2,1}u_{2,3} + u_{3,1}u_{3,3} \\ u_{1,1}u_{1,2} + u_{2,1}u_{2,2} + u_{3,1}u_{3,2} & u_{2,1}^2 + u_{2,2}^2 + u_{2,3}^2 & u_{1,2}u_{1,3} + u_{2,2}u_{2,3} + u_{3,2}u_{3,3} \\ u_{1,1}u_{1,3} + u_{2,1}u_{2,3} + u_{3,1}u_{3,3} & u_{1,2}u_{1,3} + u_{2,2}u_{2,3} + u_{3,2}u_{3,3} & u_{3,1}^2 + u_{3,2}^2 + u_{3,3}^2 \end{pmatrix}.$$

Component wise, this result in the following relations:

$$\varepsilon_{11} = u_{1,1} + \frac{1}{2}\left(u_{1,1}^2 + u_{2,1}^2 + u_{3,1}^2\right)$$

$$\varepsilon_{12} = \varepsilon_{21} = \frac{1}{2}\left(u_{2,1} + u_{1,2} + u_{1,1}u_{1,2} + u_{2,1}u_{2,2} + u_{3,1}u_{3,2}\right)$$

$$\varepsilon_{13} = \varepsilon_{31} = \frac{1}{2}\left(u_{3,1} + u_{1,3} + u_{1,1}u_{1,3} + u_{2,1}u_{2,3} + u_{3,1}u_{3,3}\right)$$

$$\varepsilon_{22} = u_{2,2} + \frac{1}{2}\left(u_{2,1}^2 + u_{2,2}^2 + u_{2,3}^2\right)$$

$$\varepsilon_{23} = \varepsilon_{32} = \frac{1}{2}\left(u_{3,2} + u_{2,3} + u_{1,2}u_{1,3} + u_{2,2}u_{2,3} + u_{3,2}u_{3,3}\right)$$

$$\varepsilon_{33} = u_{3,3} + \frac{1}{2}\left(u_{3,1}^2 + u_{3,2}^2 + u_{3,3}^2\right).$$

A.2. Nitsche's Method Derivation for the Poisson Equation

In the following, we provide a more detailed derivation of Nitsche's method for the Poisson equation.[1] We start with the strong formulation of the problem

$$-\Delta u = f \;\; in \;\; \Omega$$

$$u = g \;\; on \;\; \partial\Omega.$$

The weak from of the equation (partial integration) is

$$-\int \Delta u \, v dx = \int \nabla u \nabla v dx - \oint \nabla \, u_n \, v dc,$$

with $u \in V_h$ and $\forall v \in V_h$, where V_h is a Hilbert space. For case of simplicity, we denote the Ω domain integrals as $\int$ and the $\partial\Omega$ boundary integral as $\oint$. We use the functional $J(u)$ from [69]

$$J(u) = \int u_x^2 + u_y^2 - 2 \oint u \, (\nabla u \, n) - \psi \oint u^2,$$

$$J(u) = \int u_x^2 + u_y^2 - 2 \oint u \, \nabla u_n - \psi \oint u^2.$$

Solving the problem is nothing else than minimizing the following problem:

$$J(u - u_h) = \inf_{v \in V_h} J(u - v)$$

$$J(u-v) = \int (u_x - v_x)^2 + (u_y - v_y)^2 - 2 \oint (u-v) \, (u_n - v_n) - \psi \oint (u-v)^2$$

$$J(u-v) = \int u_x^2 + u_y^2 + v_x^2 + v_y^2 - 2(u_x v_x + u_y v_y)$$

$$-2 \oint u_n n - v_n v - u_n v + v_n v - \psi \oint u^2 + v^2 + 2uv$$

$$J(u-v) = J(u) + J(v) - 2 \int u_x v_x + u_y v_y + 2 \oint v_n u + u_n v - 2\psi \oint uv$$

$$J(u-v) = J(u) + J(v) - 2 \int u_x v_x + u_y v_y + 2 \oint v_n u + u_n v - 2\psi \oint uv$$

$$J(u-v) = J(u) + J(v) - 2 \int -u_{xx} v - u_{yy} v - 2 \oint u_n v$$

$$+2 \oint v_n u + u_n v - 2\psi \oint uv$$

[1] Special thanks to Dr. rer. nat. habil. Miriam Mehl for her help.

$$J(u-v) = J(u) + J(v) - 2\int fv + 2\oint v_n g - 2\psi \oint gv$$

$$J(u-v) = J(u) + J(v) - 2\int fv + 2\oint g(v_n - \psi v)$$

$$J(u-v) = J(u) + J(v) - 2\int fv + 2\oint g((\nabla v n) - \psi v).$$

The next step is to minimize

$$J(u-v) = J(u) + J(v) - 2\int fv + 2\oint g(v_n - \psi v)$$

$$J(u-v) = J(u) + J(v) + F(u,v).$$

Writing out this equation becomes:

$$\frac{\partial J(u-v)}{\partial v} = 0$$

$$\frac{\partial J(u-v)}{\partial v} = \frac{\partial J(v)}{\partial v} + \frac{\partial F(v,u)}{\partial v} = 0$$

$$\frac{\partial J(v)}{\partial v} = \int 2v_x v_x^{'} + \int 2v_y v_y^{'}$$

$$-2\oint(\nabla v^{'} n)v - 2\oint(\nabla v n)v^{'} + 2\psi\oint vv^{'}$$

$$\frac{\partial F(u,v)}{\partial v} = -2\int fv^{'} + 2\oint g(\nabla^{'} n) - \psi v^{'}.$$

The final equation is

$$\int v_x v_x^{'} + \int v_y v_y^{'} - \oint(\nabla v^{'} n)v - \oint(\nabla v n)v^{'} + \psi\oint vv^{'}$$

$$-\int fv^{'} + \oint g((\nabla v^{'} n) - \psi v^{'}) = 0$$

$$A = \int v_x v_x^{'} + \int v_y v_y^{'} - \oint(\nabla v^{'} n)v - \oint(\nabla v n)v^{'} + \psi\oint vv^{'}$$

$$b = \int fv^{'} - \oint(g(\nabla v^{'} n) - \psi v^{'}) = 0.$$

This gives rise to the linear problem to solve

$$A\,v = b,$$

with unknown function u and test function v

$$\int \nabla u \nabla v - \oint(\nabla u n)v - \oint u(\nabla v n) + \psi\oint uv$$

$$= \int fv - \oint g\nabla v n - \psi\oint gv.$$

A.3. Nitsche's Method Derivation for the Navier-Stokes Equations

The starting point is the strong formulation of the Stokes problem.[2] We consider only the two-dimensional case and we denote the velocity vector with (u,v) and the pressure with the scalar p:

$$-\nu\Delta(u,v)+\nabla p=(f_1,f_2)\quad in\ \ \Omega$$
$$u_x+v_y=0\ \ in\ \ \Omega$$
$$(u,v)=(g_1,g_2)\quad on\ \ \partial\Omega.$$

In the following, we derive Nitsche's method of this Stokes problem to impose the Dirichlet boundary condition (g_1,g_2) on $\partial\Omega$. Similar to the Poisson equation, we start with the definition of the energy functional $J(u,v,p)$. Next, the integrals $\int$ and $\oint$ are the domain integral $\int_\Omega dx$ and the boundary integral $\oint_{\partial\Omega} dc$ respectively:

$$J\begin{pmatrix}u\\v\\p\end{pmatrix}=\frac{\nu}{2}\int(u_x+u_y)^2-\nu\oint u_nu+\frac{\nu}{2}\int(v_x+v_y)^2-\nu\oint v_nv$$
$$-\int p(u_x+v_x)+\oint p(n_1u+n_2v),$$

where the normal vector pointing outwards of the domain is defined as $\mathbf{n}=(n_1,n_2)$. The next step is to build the difference between the exact solution (u,v,p) and the approximated solution $(\hat{u},\hat{v},\hat{p})$

$$J\left(\begin{pmatrix}u\\v\\p\end{pmatrix}-\begin{pmatrix}\hat{u}\\\hat{v}\\\hat{p}\end{pmatrix}\right)=J\begin{pmatrix}u\\v\\p\end{pmatrix}+\nu\int(u_x\hat{u}_x+u_y\hat{u}_y)+\nu\oint u_n\hat{u}+\nu\oint\hat{u}_nu$$
$$+\nu\int(v_x\hat{v}_x+v_y\hat{v}_y)+\nu\oint v_n\hat{v}+\nu\oint\hat{v}_nv$$
$$+\int p(\hat{u}_x+\hat{v}_x)+\int\hat{p}(u_x+v_y)-\oint p\mathbf{n}\cdot(u,v)^T+J\begin{pmatrix}\hat{u}\\\hat{v}\\\hat{p}\end{pmatrix}.$$

Using the BC, the continuum equation on $\partial\Omega$, and integration by parts we get the following expression

$$J\left(\begin{pmatrix}u\\v\\p\end{pmatrix}-\begin{pmatrix}\hat{u}\\\hat{v}\\\hat{p}\end{pmatrix}\right)=J\begin{pmatrix}u\\v\\p\end{pmatrix}+\int(\nu\Delta u-p_x)\hat{u}+\int(\nu\Delta v-p_y)\hat{v}$$

[2] Special thanks to Dr. rer. nat. habil. Miriam Mehl for her help

$$+\nu \oint (\hat{u}_n g_1 + \hat{v}_n g_2) - \oint \hat{p}\mathbf{n} \cdot (g_1, g_2)^T + J \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}.$$

We notice, that the second and the third terms form the momentum equation that we use again to further simplify the expression

$$J\left(\begin{pmatrix} u \\ v \\ p \end{pmatrix} - \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}\right) = J \begin{pmatrix} u \\ v \\ p \end{pmatrix} + \int (f_1, f_2) \cdot (\hat{u}, \hat{v})^T$$

$$+ \oint (\nu (\hat{u}_n, \hat{v}_n) - \hat{p}\mathbf{n}) \cdot (g_1, g_2)^T + J \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}.$$

This gives rise to the minimization problem that results in Nitsche's formula for the Stokes equations, with the discrete space V_h for the velocity and P_h for the pressure

$$J\left(\begin{pmatrix} u \\ v \\ p \end{pmatrix} - \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}\right) = \min_{(\tilde{u},\tilde{v},\tilde{p}) \in V_h \times V_h \times P_h} J\left(\begin{pmatrix} u \\ v \\ p \end{pmatrix} - \begin{pmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{p} \end{pmatrix}\right) \tag{A.5}$$

In the next step, we introduce the basis functions for $\hat{u}$, $\hat{v}$ and $\hat{p}$ such that the minimization problem (A.5) can be solved for the discrete space.

$$\hat{u} = \sum_{i=1}^{E} \alpha_i \phi_1^1, \quad \hat{v} = \sum_{i=1}^{E} \beta_i \phi_1^2, \quad \hat{p} = \sum_{i=1}^{F} \delta_i \sigma_1.$$

This results for $\hat{u}$

$$\frac{\partial}{\partial \alpha_i} J\left(\begin{pmatrix} u \\ v \\ p \end{pmatrix} - \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}\right) = \mu \sum_{i=j}^{E} \alpha_i \int \left(\phi_{i,x}^1 \phi_{j,x}^1 + \phi_{i,y}^1 \phi_{j,y}^1\right) - \sum_{j=1}^{F} \delta_j \int \sigma_j \phi_{i,x}^1$$

$$- \sum_{j=1}^{E} \alpha_j \int \phi_{i,n}^1 \phi_j^1 - \sum_{j=1}^{E} \alpha_j \int \phi_i^1 \phi_{j,n}^1 + \sum_{j=1}^{F} \delta_j \oint \sigma_j n_1 \phi_i^1 + \nu \oint \phi_{i,n}^1 g_1 - \int f_1 \phi_i^1.$$

Similar for $\hat{v}$

$$\frac{\partial}{\partial \beta_i} J\left(\begin{pmatrix} u \\ v \\ p \end{pmatrix} - \begin{pmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{pmatrix}\right) = \mu \sum_{i=j}^{E} \beta_i \int \left(\phi_{i,x}^2 \phi_{j,x}^2 + \phi_{i,y}^2 \phi_{j,y}^2\right) - \sum_{j=1}^{F} \delta_j \int \sigma_j \phi_{i,x}^2$$

$$- \sum_{j=1}^{E} \beta_j \int \phi_{i,n}^2 \phi_j^2 - \sum_{j=1}^{E} \beta_j \int \phi_i^2 \phi_{j,n}^2 + \sum_{j=1}^{F} \delta_j \oint \sigma_j n_2 \phi_i^2 + \nu \oint \phi_{i,n}^2 g_2 - \int f_2 \phi_i^2.$$

Finally, for $\hat{p}$

$$\frac{\partial}{\partial \delta_i} J\left(\left(\begin{array}{c} u \\ v \\ p \end{array}\right) - \left(\begin{array}{c} \hat{u} \\ \hat{v} \\ \hat{p} \end{array}\right)\right) = -\sum_{j=1}^{N} \int \sigma_i \left(\alpha_j \phi^1_{j,x} + \beta_j \phi^2_{j,y}\right)$$

$$+\sum_{j=1}^{N} \oint \sigma_i \left(\alpha_j n_1 \phi^1_j + \beta_j n_2 \phi^2_j\right) - \oint \sigma_i \left(n_1 g_1 + n_2 g_2\right).$$

Summing up the equations above results in the following operators

$$a\left(\hat{u}, \hat{v}, \hat{p}\right)\left(\phi^1, \phi^2, \sigma\right) := \mu \int \left(u_x \phi^1_x + u_y \phi^2_y\right) + \mu \int \left(v_x \phi^2_x + v_y \phi^2_y\right)$$

$$-\int \hat{p}\left(\phi^1_x + \phi^2_y\right) - \int \sigma\left(u_x + v_y\right),$$

$$b\left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2, \sigma\right) := \mu \oint \left(\phi^1_n \hat{u} + \phi^2_n \hat{v}\right) - \oint \sigma\left(n_1 \hat{u} + n_2 \hat{v}\right),$$

$$c\left(\hat{u}, \hat{v}, \hat{p}\right)\left(\phi^1, \phi^2, \sigma\right) := \mu \oint \left(\phi^1 \hat{u}_n + \phi^2 \hat{v}_n\right) - \oint \hat{p}\left(n_1 \phi^1 + n_1 \phi^2\right),$$

$$f\left(\phi^1, \phi^2, \sigma\right) := \int \left(f_1 \phi^1 + f_2 \phi^2\right).$$

In order to make the resulting problem a positive definite, we extend the operator $b\left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2, \sigma\right)$ with additional stabilization terms

$$\hat{b}\left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2, \sigma\right) := b\left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2, \sigma\right) + \mu \frac{\gamma_1}{h} \oint \left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2\right)^T$$

$$+\frac{\gamma_2}{h} \oint \left(n_1 \hat{u}, n_2 \hat{v}\right)\left(n_1 \phi^1, n_2 \phi^2\right)^T,$$

where γ_1 and γ_2 are the penalty coefficients, and h represents the mesh width on $\partial\Omega$. With this modified operator, the final form of Nitsche's method result that imposes the Dirichlet boundary condition (g_1, g_2) for the presented two-dimensional Stokes problem,

$$a\left(\hat{u}, \hat{v}, \hat{p}\right)\left(\phi^1, \phi^2, \sigma\right) - \hat{b}\left(\hat{u}, \hat{v}\right)\left(\phi^1, \phi^2, \sigma\right) - c\left(\hat{u}, \hat{v}, \hat{p}\right)\left(\phi^1, \phi^2, \sigma\right) =$$

$$-\hat{b}\left(g_1, g_2\right)\left(\phi^1, \phi^2, \sigma\right) + f\left(\phi^1, \phi^2, \sigma\right). \tag{A.6}$$

(A.6) is the resulting Nitsche's method for the Stokes equations in 2D that also can be extended for 3D, and in the same form can be used for the Navier-Stokes equations. This formula is also listed in [15, 41] for the Navier-Stokes equations.

A.4. Sundance Code for the Navier-Stokes Equations with Nitsche's Method

Sundance code of the 2D stationary Navier-Stokes equations with Nitsche's method. (ux, uy) and (vx, vy) are the velocity unknown and test funcitons. p and q are the pressure field's unknown and test function. The expression eqn represents Nitsche's method.

```
ParametrizedCurve curve = new Polygon2D("polygon.txt",1.0,1e-7);
ParametrizedCurve curveIntegral = new ParamCurveIntegral(curve);
Expr nu = new Sundance::Parameter(1.0/1000.0);
Expr h = new CellDiameterExpr();
Expr nx = new CurveNormExpr(0);
Expr ny = new CurveNormExpr(1);
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
QuadratureFamily quad_hi = new GaussLobattoQuadrature(6);
QuadratureFamily quad_c12 = new GaussianQuadrature(12);
QuadratureFamily quad_c = new PolygonQuadrature( quad_c12 );
Expr eqn = Integral(OutsideCurve, nu*(grad*vx)*(grad*ux)
    + nu*(grad*vy)*(grad*uy)
    + vx*(ux*(dx*ux)+uy*(dy*ux))+ vy*(ux*(dx*uy)+uy*(dy*uy))
    - p*(dx*vx+dy*vy) + q*(dx*ux+dy*uy), quad4)
  + Integral(OnCurve, nu*(grad*vx)*(grad*ux)
    + nu*(grad*vy)*(grad*uy)
    + vx*(ux*(dx*ux)+uy*(dy*ux)) + vy*(ux*(dx*uy)+uy*(dy*uy))
    - p*(dx*vx+dy*vy) + q*(dx*ux+dy*uy), quad_hi , curve)
  + Integral(OnCurve,  -nu*(dx*ux*nx+dy*ux*ny)*vx
    - nu*(dx*uy*nx+dy*uy*ny)*vy + p*(nx*vx+ny*vy)
    - nu*(dx*vx*nx+dy*vx*ny)*(ux-ux_Dirichlet)
    - nu*(dx*vy*nx+dy*vy*ny)*(uy-uy_Dirichlet)
    - q*(nx*(ux-ux_Dirichlet)+ny*(uy-uy_Dirichlet))
    + nu*ga1/h*((ux-ux_Dirichlet)*vx+(uy-uy_Dirichlet)*vy)
    + ga2/h*((ux-ux_Dirichlet)*nx+(uy-uy_Dirichlet)*ny)*(vx*nx+vy*ny)
      quad_c , curveIntegral );
//...the solution is computed and is in the up[0],up[1],up[2] spaces
Expr dragExpr = Integral( OnCurve ,
    -rho*nu*ny*(nx*dx*(ny*up[0]-nx*up[1])+ny*dy*(ny*up[0]-nx*up[1]))
    + nx*up[2], quad_c , curveIntegral );
FunctionalEvaluator dragInt( mesh , dragExpr);
double dragIntVal = dragInt.evaluate();
```

```
Expr liftExpr = Integral( OnCurve ,
     rho*nu*nx*( nx*dx*(ny*up[0]-nx*up[1])+ny*dy*(ny*up[0]-nx*up[1]))
   + ny*up[2], quad_c , curveIntegral );
FunctionalEvaluator liftInt( mesh , liftExpr);
double liftIntVal = liftInt.evaluate();
```

Sundance code of the 2D transient Navier-Stokes equations with Nitsche's method. (ux, uy) and (vx, vy) are the velocity unknown and test funcitons. p and q are the pressure field's unknown and test function.

```
Expr upo = new DiscreteFunction(VelPreSpace, 0.0, "upo");
Expr up = new DiscreteFunction(VelPreSpace, 0.0, "up");
Expr umx=0.5*(ux+upo[0]);
Expr umy=0.5*(uy+upo[1]);
Expr dxumx=0.5*(dx*ux+dx*upo[0]);
Expr dxumy=0.5*(dx*uy+dx*upo[1]);
Expr dyumx=0.5*(dy*ux+dy*upo[0]);
Expr dyumy=0.5*(dy*uy+dy*upo[1]);
Expr pm=0.5*(p+upo[2]);
double dt=1e-4;
Expr bc = EssentialBC(walls + left,(1.0/h)*v*u,quad4);
Expr eqn = Integral(OutsideCurve,
   (1.0/dt)*(ux*vx+uy*vy-upo[0]*vx-upo[1]*vy)
   + nu*((dx*vx)*dxumx+(dy*vx)*dyumx)
   + nu*((dx*vy)*dxumy+(dy*vy)*dyumy)
   + vx*(umx*dxumx+umy*dyumx) + vy*(umx*dxumy+umy*dyumy)
   - p*(dx*vx+dy*vy) + q*(dx*ux+dy*uy), quad4)
+ Integral(OnCurve,
   (1.0/dt)*(ux*vx+uy*vy-upo[0]*vx-upo[1]*vy)
   + nu*((dx*vx)*dxumx+(dy*vx)*dyumx)
   + nu*((dx*vy)*dxumy+(dy*vy)*dyumy)
   + vx*(umx*dxumx+umy*dyumx) + vy*(umx*dxumy+umy*dyumy)
   - p*(dx*vx+dy*vy) + q*(dx*ux+dy*uy),quad_hi , curve)
+ Integral(OnCurve, -nu*(dxumx*nx+dyumx*ny)*vx
   -nu*(dxumy*nx+dyumy*ny)*vy + p*(nx*vx+ny*vy)
   -nu*(dx*vx*nx+dy*vx*ny)*(umx-ux_Dirichlet)
   -nu*(dx*vy*nx+dy*vy*ny)*(umy-uy_Dirichlet)
   -q*(nx*(ux-ux_Dirichlet)+ny*(uy-uy_Dirichlet))
   +nu*ga1/h*((ux-ux_Dirichlet)*vx+(uy-uy_Dirichlet)*vy)
   +ga2/h*((ux-ux_Dirichlet)*nx+(uy-uy_Dirichlet)*ny)*(vx*nx+vy*ny),
    quad_c , curveIntegral );
NonlinearProblem prob(mesh, eqn , bc, List(vx,vy,q),
                         List(ux,uy,p), up, vecType );
```

```
// time stepping until 8.0 seconds by dt
for (double t=dt ; t<=8.0; t = t + dt) {
     NOX::StatusTest::StatusType status = prob.solve(solver);
     // compute the lift and drag coeffcients
     double c_D = 2.0*dragInt.evaluate()/(Umean*Umean*D);
     double c_L = 2.0*liftInt.evaluate()/(Umean*Umean*D);
     CopyDiscreteFunction(upo,up);
}
```

Sundance code of the 3D stationary Navier-Stokes equations with Nitsche's method. (ux, uy, uz) and (vx, vy, vz) are the velocity unknown and test funcitons. p and q are the pressure field's unknown and test function.

```
QuadratureFamily quad_curve = new SurfQuadrature(quad_gauss);
QuadratureFamily quad_hi = new GaussLobattoQuadrature(6);
Expr nx = new CurveNormExpr(0);
Expr ny = new CurveNormExpr(1);
Expr nz = new CurveNormExpr(2);
Expr eqn = Integral(OutsideCircle, nu*(grad*vx)*(grad*ux)
    + nu*(grad*vy)*(grad*uy) + nu*(grad*vz)*(grad*uz)
    + vx*(u*grad)*ux + vy*(u*grad)*uy + vz*(u*grad)*uz
    - (1/rho_fluid)*p*(dx1*vx+dx2*vy+dx3*vz)
    + (dx1*ux+dx2*uy+dx3*uz)*q , quad4)
+ Integral(OnCircle, nu*(grad*vx)*(grad*ux)
    + nu*(grad*vy)*(grad*uy) + nu*(grad*vz)*(grad*uz)
    + vx*(u*grad)*ux + vy*(u*grad)*uy + vz*(u*grad)*uz
    - (1/rho_fluid)*p*(dx1*vx+dx2*vy+dx3*vz)
    + (dx1*ux+dx2*uy+dx3*uz)*q , quad_hi , curve )
+ Integral( OnCircle , -nu*(dx1*ux*nx+dx2*ux*ny+dx3*ux*nz)*vx
 - nu*(dx1*uy*nx+dx2*uy*ny+dx3*uy*nz)*vy
 - nu*(dx1*uz*nx+dx2*uz*ny+dx3*uz*nz)*vz
 - nu*(dx1*vx*nx+dx2*vx*ny+dx3*vx*nz)*(ux-ux_Dirichlet)
 - nu*(dx1*vy*nx+dx2*vy*ny+dx3*vy*nz)*(uy-uy_Dirichlet)
 - nu*(dx1*vz*nx+dx2*vz*ny+dx3*vz*nz)*(uz-uz_Dirichlet)
 + (1/rho_fluid)*p*(nx*vx+ny*vy+nz*vz)
 - q*(nx*(ux-ux_Dirichlet)+ny*(uy-uy_Dirichlet)+nz*(uz-uz_Dirichlet))
 + nu*ga1/h*((ux-ux_Dirichlet)*vx+(uy-uy_Dirichlet)*vy
   +(uz-uz_Dirichlet)*vz)
 + ga2/h*((ux-ux_Dirichlet)*nx+(uy-uy_Dirichlet)*ny+
 (uz-uz_Dirichlet)*nz)*(vx*nx+vy*ny+vz*nz),quad_curve,curveIntegral)
// the solution is in up_f[0...3] spaces, compute the X,Y and Z forces
Expr ForceX = Integral(OnCircle,up_f[3]*nx
  - rho_fluid*nu*( 2.0*nx*(dx1*up_f[0]) + ny*(dx1*up_f[1]+dx2*up_f[0])
```

```
  + nz*(dx1*up_f[2]+dx3*up_f[0])),quad_curve,curveIntegral);
FunctionalEvaluator ForceX_V(mesh,ForceX);
Expr ForceY = Integral( OnCircle,up_f[3]*ny
  - rho_fluid*nu*( 2.0*ny*(dx2*up_f[1]) + nx*(dx2*up_f[0]+dx1*up_f[1])
  + nz*(dx2*up_f[2]+dx3*up_f[1])),quad_curve,curveIntegral);
FunctionalEvaluator ForceY_V( mesh , ForceY );
Expr ForceZ = Integral(OnCircle,up_f[3]*nz
  - rho_fluid*nu*( 2.0*nz*(dx3*up_f[2]) + nx*(dx1*up_f[2]+dx3*up_f[0])
  + ny*(dx2*up_f[2]+dx3*up_f[1])), quad_curve,curveIntegral);
FunctionalEvaluator ForceZ_V( mesh , ForceZ );
```

A.5. Sundance Code for Static Partitioned FSI Computations

In the following, we consider one Sundance code for the partitioned computation of a sationary FSI problem in 2D. *prob_s* represents the structure problem, whereas *prob_fl* represents the fluid problem. fx and fy represent the coupling stress vectors and are used accordingly in *eqn_s* to couple them to the structure's problem.

```
ParametrizedCurve curve_fl=new Polygon2D("FSI_polygon.txt",1.0,1e-7);
curve_fl.addNewScalarField( "dispX" , 0.0 );
curve_fl.addNewScalarField( "dispY" , 0.0 );
curve_fl.addNewScalarField( "fx" , 0.0 );
curve_fl.addNewScalarField( "fy" , 0.0 );
ParametrizedCurve curve_str = polyg->createTwinPolygon(0,0,1,1);
...
Expr fx = new UserDefOp(List(x1,x2), rcp(new CurveExpr(curve_str,2)) );
Expr fy = new UserDefOp(List(x1,x2), rcp(new CurveExpr(curve_str,3)) );
Expr eqn_s = Integral( Omega , LduT*C*( Lnl * us ) , quad4 ) +
  Integral(OnCurve_S,-fx*du[0]-fy*du[1],quad_c,curveIntegral_str);
...
Array<double>& dispX = curve_str.getScalarFieldValues(0);
Array<double>& dispY = curve_str.getScalarFieldValues(1);
Array<Point>& pnt_f = curve_fl.getControlPoints();
Array<Point>& pnt_s = curve_str.getControlPoints();
for( iter = 0 ; error > 1e-6 ; iter++)
{
    double dist = 0.0 , w = 0.31;
    for (int p = 0 ; p < pnt_f.size() ; p++ ){
     Point p_tmp = pnt_f[p];
     pnt_f[p][0]=pnt_f[p][0]-w*(pnt_f[p][0] - pnt_s[p][0] - dispX[p]);
```

```
     pnt_f[p][1]=pnt_f[p][1]-w*(pnt_f[p][1] - pnt_s[p][1] - dispY[p]);
     dist = dist + (pnt_f[p]-p_tmp)*(pnt_f[p]-p_tmp);
    }
    error = sqrt(dist/(double)pnt_f.size());
    curve_fl.update();
    // trigger the reassamle of both problem
    prob_s.reAssembleProblem();
    prob_fl.reAssembleProblem();
    // Solve the fluid system
    status_fl = prob_fl.solve(solver_fl);
    // set the drag and lift forces
    curve_fl.setSpaceValues( dragInt_Curve , 2 );
    curve_fl.setSpaceValues( liftInt_Curve , 3 );
    // Solve the structure system
    status_s = prob_s.solve(solver_s);
    // set the X and Y displacements
    curve_str.setSpaceValues( dispXInt_Curve , 0 );
    curve_str.setSpaceValues( dispYInt_Curve , 1 );
}
```

Bibliography

[1] M. Abramowitz and I.A. Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables.* Number Bd. 55,Nr. 1972 in Applied mathematics series. 1964.

[2] P. Angot, C.-H. Bruneau, and P. Fabrie. A penalization method to take into account obstacles in incompressible viscous flows. *Numerische Mathematik*, 81(4):497–520, 1999.

[3] I. Babuška. The finite element method with lagrangian multipliers. *Numerische Mathematik*, 20:179–192, 1973.

[4] I. Babuška, U. Banerjee, and J.E. Osborn. *Meshless and Generalized Finite Element Methods: A Survey of Some Major Results, in Meshfree Methods for Partial Differential Equations.* Lect. Notes Comput. Sci. Eng. 26 pp. 1-20. Springer, New York, 2002.

[5] M. Bader, S. Schraufstetter, C. A. Vigh, and Jörn Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using sierpinski curves. 4(1):12–21, November 2008.

[6] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic finite element codes. *to appear in ACM Trans. Math. Software.*

[7] W. Bangerth, R. Hartmann, and Kanschat G. deal.ii – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):1–27, 2007.

[8] W. Bangerth and G. Kanschat. Concepts for object-oriented finite element software – the `deal.II` library. Preprint 99-43 (SFB 359), IWR Heidelberg, October 1999.

[9] W. Bangerth and O. Kayser-Herold. Data structures and requirements for hp finite element software. *ACM Trans. Math. Softw.*, 36:4:1–4:31, March 2009.

[10] I. W. Barbara. A mortar finite element method using dual spaces for the lagrange multiplier. *SIAM J. Numer. Anal*, 38:989–1012, 1998.

[11] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive

scientific computing. part ii: implementation and tests in dune. *Computing*, 82:121–138, 2008.

[12] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework. *Computing*, 82:103–119, 2008.

[13] Y. Bazilevs and T.J.R. Hughes. Weak imposition of dirichlet boundary conditions in fluid mechanics. *Computers and Fluids*, 36(1):12 – 26, 2007.

[14] Y. Bazilevs, C. Michler, V.M. Calo, and T.J.R. Hughes. Weak dirichlet boundary conditions for wall-bounded turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 196(49-52):4853 – 4862, 2007.

[15] R. Becker. Mesh adaptation for Dirichlet flow control via Nitsche's method. *Communications in Numerical Methods in Engineering*, 18(9):669–680, 2002.

[16] R. Becker, M. Braack, R. Rannacher, and C. Waguet. Fast and reliable solutions of the navier-stokes equations including chemistry. *Computer and Visualization in Science*, 2(3), 1999.

[17] T. Belytschko and T. Black. Elastic crack growth in finte elements with minimal remeshing. *International Journal for Numerical Methods in Engineering*, 45:601–620, 1999.

[18] J. Benk, R. Kirby, K. Long, and M. Mehl. Adaptive parallel cartesian mesh in a fem pde-toolbox environment. *ACM Trans. Math. Softw.*, (submitted):0, January 2012.

[19] J. Benk, M. Mehl, and M. Ulbrich. Sundance pde solvers on cartesian fixed grids in complex and variable geometries. In *Proceedinggs of the ECCOMAS Thematic Conference CFD & Optimization, Antlya, Turkey, May 23-25, 2011*, 2011.

[20] L. Biros, G. Ying and Zorin D. The embedded boundary integral method (ebi) for the incompressible navier-stokes equations, 2002.

[21] D. Braess. *Finite elements: theory, fast solvers, and applications in elasticity theory.* Cambridge University Press, 2007.

[22] M. Brenk. *Algorithmic Aspects of Fluid-Structure Interactions on Cartesian Grids (German: Algorithmische Aspekte der Fluid-Struktur-Wechselwirkung auf kartesischen Gittern).* PhD thesis, Technische Universität München, 2007.

[23] S.C. Brenner and L.R. Scott. *The mathematical theory of finite element methods.* Texts in applied mathematics. Springer-Verlag, 2002.

[24] H. Brinkman. A calculation of the viscous force exerted by a flowing fluid on a

dense swarm of particles. *Applied Scientific Research*, 1:27–34, 1949.

[25] H.-J. Bungartz, J. Benk, B. Gatzhammer, M. Mehl, and T. Neckel. *Fluid-Structure Interaction – Modelling, Simulation, Optimisation, Part II*, volume 73 of *LNCSE*, chapter Partitioned Simulation of Fluid-Structure Interaction on Cartesian Grids, pages 255–284. Springer, Berlin, Heidelberg, October 2010.

[26] H.-J. Bungartz, B. Gatzhammer, M. Lieb, M. Mehl, and T. Neckel. Towards multiphase flow simulations in the pde framework peano. *Computational Mechanics*, 48(3):365–376, 2011.

[27] C. Burstedde, L.C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[28] R. Codina and J. Baiges. Approximate imposition of boundary conditions in immersed boundary methods. *International Journal for Numerical Methods in Engineering*, 80(11):1379–1405, 2009.

[29] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module. *Computing*, 90(3–4):165–196, 2010.

[30] J. Degroote. *Development of algorithms for the partitioned simulation of strongly coupled fluid-structure interaction problems.* Dissertation, Ghent University. Faculty of Engineering, 2010.

[31] Th. Dunne, R. Rannacher, and Th. Richter. Numerical simulation of fluid-structure interaction based on monolithic variational formulations. *Comtemporary Challenges in Mathematical Fluid Mechanics (G.P. Galdi, R. Rannacher, eds.), World Scientific, Singapore*, 2010.

[32] ExodusII. http://sourceforge.net/projects/exodusii/, 2007.

[33] FEniCS. http://fenicsproject.org, 2012.

[34] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[35] A. Gerstenberger. *An XFEM based fixed-grid approach to fluid-structure interaction.* Dissertation, Technische Universität München, 2010.

[36] M. Giles, M. G. Larson, J. M. Levenstam, and E. Süli. Adaptive error control for finite element approximations of the lift and drag coefficients in viscous flow. Technical Report NA-97/06, Oxford University Computing Laboratory, 1997.

[37] R. Glowinski, T. W. Pan, T. I. Helsa, D. D. Joseph, and J. Périaux. A fictitious domain approach to the direct numerical simulation of incompressible viscous flow past moving rigid bodies: application to particulate flow. *J. Comput. Phys.*, 169(2):363–426, 2001.

[38] P. M. Gresho, R. L. Sani, and M. S. Engelman. *Incompressible Flow and the Finite Element Method*. John Wiley & Sons, 1998.

[39] A. Hansbo and P. Hansbo. A finite element method for the simulation of strong and weak discontinuities in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 193(33-35):3523 – 3540, 2004.

[40] P. Hansbo. Nitsche's method for interface problems in computational mechanics. *Chimera*, 1(2):1–27, 2005.

[41] P. Hansbo and M. Juntunen. Weakly imposed dirichlet boundary conditions for the brinkman model of porous media flow. *Applied Numerical Mathematics*, 59(9):1274–1289, 2009.

[42] J. Haslinger, J.-F. Maitre, and L. Tomas. Fictitious domain methods with distributed lagrange multipliers part i: Application to the solution of elliptic state problems. *Mathematical Models and Methods in Applied Sciences*, 11(3):521–547, 2001.

[43] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and Williams A. An overview of trilinos. Technical report, Sandia National Laboratories, 2003.

[44] C. Hinterberger and M. Olesen. Automatic geometry optimization of exhaust systems based on sensitivities computed by a continuous adjoint cfd method in openfoam. *SAE Library*, 2010.

[45] C. W. Hirt, A.-A. Amsden, and J. L. Cook. An arbitrary lagrangian-eulerian computing method for all flow speeds. *J. Comp. Phys.*, 14:227–253, 1974.

[46] J. Hron and S. Turek. Proposal for numerical benchmarking of fluid-structure interaction between elastic object and laminar incompressible flow. In H.-J. Bungartz and M. Schäfer, editors, *Fluid-Structure Interaction*, number 53 in Lecture Notes in Computational Science and Engineering, pages 371–385. Springer-Verlag, 2006.

[47] A. Huerta, T. Belytschko, T. Fernandez-Mendez, and T. Rabczuk. *Meshfree Methods*. vol. 1 of Encyclopedia of Computational Mechanics ch. 10, pp. 279-309. Wiley, 2004.

[48] T.J.R. Hughes, G. Scovazzi, and L.P. Franca. *Multiscale and Stabilized Methods*. in Encyclopedia of Computational Mechanics , eds. E. Stein, R. De Borst, T. J. R.

Hughes. Wiley, 2004.

[49] M. Juntunen and R. Stenberg. Nitsche's method for general boundary conditions. *Math. Comp*, 78(267):1353–1374, 2009.

[50] H. Kardestuncer, D. H. Norrie, and Brezzi F. *Finite element handbook*. McGraw-Hill reference books of interest: Handbooks. McGraw-Hill, 1987.

[51] K. Khadra, P. Angot, S. Parneix, and J.-P. Caltagirone. Fictitious domain approach for numerical modelling of navier-stokes equations. *International Journal for Numerical Methods in Fluids*, 34(8):651–684, 2000.

[52] B. Khalighi, S. Jindal, J.P. Johnson, K.H. Chen, and G. Iaccarino. Validation of the immersed boundary cfd approach for complex aerodynamic flows. In Fred Browand, Rose McCallen, and James Ross, editors, *The Aerodynamics of Heavy Vehicles II: Trucks, Buses, and Trains*, volume 41 of *Lecture Notes in Applied and Computational Mechanics*, pages 21–38. Springer Berlin / Heidelberg, 2009.

[53] R.C. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3), 2006.

[54] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006.

[55] U. Kuettler and W.A. Wall. Fixed-point fluid-structure interaction solvers with dynamic relaxation. In *Computational Mechanics*. Springer, 2008.

[56] M.-C. Lai and C. S. Peskin. An Immersed Boundary Method with Formal Second-Order Accuracy and Reduced Numerical Viscosity. *Journal of Computational Physics*, 160(2):705–719, May 2000.

[57] X. S. Li and J. W. Demmel. Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29:110–140, June 2003.

[58] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283–295, 2009.

[59] A. Logg, K.-A. Mardal, G.N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[60] K. Long. Sundance 2.0 tutorial, 2004.

[61] K. Long. http://www.math.ttu.edu/ klong/sundance/html/index.html, 2007.

[62] K. Long, R. Kirby, and B. van Bloemen Waanders. Unified embedded parallel finite element computations via software-based frechet differentiation. *Siam Journal on*

Scientific Computing (SISC), November 2010.

[63] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, 1987.

[64] U. Mayer, A. Popp, A. Gerstenberger, and W. Wall. 3d fluid-structure-contact interaction based on a combined xfem fsi and dual mortar contact approach. *Computational Mechanics*, 46:53–67, 2010.

[65] R.J. Meyers, T.J. Tautges, and Tuchinsky P.M. The hex-tet hex-dominant meshing algorithm as implemented in cubit. In *Proceedings of the 7th International Meshing Roundta-ble*, 1998.

[66] R. Mittal, C. Bonilla, and H.S. Udaykumar. Cartesian grid methods for simulating flows with moving boundaries. In *Computational Methods and Experimental Measurements XI, Greece*, 2003.

[67] R. Mittal and G. Iaccarino. Immersed boundary methods. *Annual Review of Fluid Mechanics*, 37(1):239–261, 2005.

[68] N. Moës, J. Dolbow, and M. Tourbieree. A finite element method for crack growth without remeshing. *International Journal for Numerical Methods in Engineering*, 46:131–150, 1999.

[69] J. Nitsche. Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 36:9–15, 1971.

[70] S. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[71] Paraview. http://www.paraview.org/, 2000.

[72] J. Parvizian, A. Düster, and E. Rank. Finite cell method: h- and p- extension for embedded domain methods in solid mechanics. *Computational Mechanics*, (41):121–133, 2007.

[73] F. Pellegrini. Scotch. url http://www.labri.fr/perso/pelegrin/scotch.

[74] C. Peskin. Flow patterns around heart valves: A numerical method. *Journal of Computational Physics*, 10(2):252–271, October 1972.

[75] P. Popov, L. Bi, Y. Efendiev, R.E. Ewing, G. Qin., J. Li, and Y. Ren. Multi-physics and multi-scale methods for modeling fluid flow through naturally-fractured vuggy carbonate reservoirs. In *SPE Middle East Oil and Gas Show and Conference*, 2007.

[76] I. Ramière, P. Angot, and M. Belliard. A fictitious domain approach with spread interface for elliptic problems with general boundary conditions. *Computer Methods in Applied Mechanics and Engineering*, 196(4-6):766 – 781, 2007.

[77] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, 2005.

[78] R. Sampath and G. Biros. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM Journal on Scientific Computing*, 32:1361–1392, 2010.

[79] B. Satish, D. G. William, C. M. Lois, and F. S. Barry. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[80] M. Schäfer and S. Turek. Benchmark computations of laminar flow around a cylinder. In *Flow simulation with high-performance computers. Bd. 2.*, volume 52 of *Notes on numerical fluid mechanics*, pages 547–566. Vieweg, Braunschweig, January 1996.

[81] D. Schillinger, M. Ruess, N. Zander, Y. Bazilevs, A. Düster, and E. Rank. Large deformation analysis with the p- and b-spline versions of the finite cell method (1) part i: A geometrically nonlinear fcm formulation based on repeated deformation resetting in the fictitious domain. *submitted to Computational Mechanics, 2011.*

[82] D. Schillinger, M. Ruess, N. Zander, Y. Bazilevs, A. Düster, and E. Rank. Large deformation analysis with the p- and b-spline versions of the finite cell method (2) part ii: Unfitted dirichlet boundary conditions, severe mesh distortion and application to complex voxel-based geometries. *submitted to Computational Mechanics, 2011.*

[83] J. R. Shewchuk. http://www.cs.cmu.edu/ quake/showme.html, 2007.

[84] R. Stenberg. On some techniques for approximating boundary conditions in the finite element method. *J. Comput. Appl. Math.*, 63:139–148, November 1995.

[85] T. Strouboulis, K. Copps, and I. M. Babuška. The generalized finite element method: an example of its implementation and illustration of its performance. *International Journal for Numerical Methods in Engineering*, 47:1401–1417, 2000.

[86] R.S. Sundar, H. Sampath and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30:2675–2708, 2008.

[87] T. E. Tezduyar and Y. Osawa. Finite element stabilization parameters computed from element matrices and vectors. *Computer Methods in Applied Mechanics and Engineering*, 190(31):411–430, 2000.

[88] S. Turek, J. Hron, M. Razzaq, H. Wobker, and M. Schäfer. *Numerical Benchmarking of Fluid-Structure Interaction: A Comparison of Different Discretization and Solution Approaches*, volume 73 of *Lecture Notes in Computational Science and Engineering*, chapter 15, pages 413–424. Springer Berlin Heidelberg, 2010.

[89] R. Verzicco, P. Orlandi, J. Mohd-Yusof, and D. Haworth. Les in complex geometries using boundary body forces. *AIAA Journal*, 38:427–433, 2000.

[90] W. Wall, A. Gerstenberger, P. Gamnitzer, C. Förster, and E. Ramm. Large deformation fluid-structure interaction - advances in ale methods and new fixed grid approaches. In Hans-Joachim Bungartz and Michael Schäfer, editors, *Fluid-Structure Interaction*, volume 53 of *Lecture Notes in Computational Science and Engineering*, pages 195–232. Springer Berlin Heidelberg.

[91] W. A. Wall. Introduction to finite elements, lecture script, 2010.

[92] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut, 2009.

[93] P. Wriggers. *Nonlinear Finite Element Methods*. Springer, 2010.

[94] C. S. Zender. Analysis of self-describing gridded geoscience data with netcdf operators (nco). *Environ. Modell. Softw.*, 23(10), 2008.

[95] A. Zilian and A. Legay. The enriched space-time finite element method (est) for simultaneous solution of fluid-structure interaction. *International Journal for Numerical Methods in Engineering*, 75(3):305–334, 2008.