Geometry–Aware Finite Element Framework for Multi–Physics Simulations

An Algorithmic and Software-Centric Perspective

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Patrick Zulian

under the supervision of
Rolf Krause

June 2017

Dissertation Committee

Kai Hormann	Università della Svizzera italiana
Illia Horenko	Università della Svizzera italiana
Christian Hesch	Universtät Siegen
Fabian Kuhn	University of Freiburg

Dissertation accepted on 30 June 2017

Research Advisor	PhD Program Director
Rolf Krause	**Michael Bronstein**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Patrick Zulian
Lugano, 30 June 2017

Dedicated to my wife Giorgiana, family, and friends.

"In mathematics you don't
understand things, you just get
used to them"

John von Neumann

Abstract

In finite element simulations, the handling of geometrical objects and their discrete representation is a critical aspect in both serial and parallel scientific software environments. The development of codes targeting such envinronments is subject to great development effort and man-hours invested. In this thesis we approach these issues from three fronts.

First, stable and efficient techniques for the transfer of discrete fields between non matching volume or surface meshes are an essential ingredient for the discretization and numerical solution of coupled multi-physics and multi-scale problems. In particular L^2-projections allow for the transfer of discrete fields between unstructured meshes, both in the volume and on the surface. We present an algorithm for parallelizing the assembly of the L^2-transfer operator for unstructured meshes which are arbitrarily distributed among different processes. The algorithm requires no a priori information on the geometrical relationship between the different meshes.

Second, the geometric representation is often a limiting factor which imposes a trade-off between how accurately the shape is described, and what methods can be employed for solving a system of differential equations. Parametric finite-elements and bijective mappings between polygons or polyhedra allow us to flexibly construct finite element discretizations with arbitrary resolutions without sacrificing the accuracy of the shape description. Such flexibility allows employing state-of-the-art techniques, such as geometric multigrid methods, on meshes with almost any shape.

Last, the way numerical techniques are represented in software libraries and approached from a development perspective affect both usability and maintainability of such libraries. Completely separating the intent of high-level routines from the actual implementation and technologies allows for portable and maintainable performance. We provide an overview on current trends in the development of scientific software and showcase our open-source library UTOPIA.

Acknowledgements

The author would like to thank the people and institutions that have been integral to the realization of this work. Prof. Dr. Rolf Krause for his advise and support during my PhD studies. Teseo Schneider for his help and collaboration in realizing the project which embodies Chapter 4, and for constant exchange of ideas and discussions. Dr. Lea Conen for her contributions for a help in the writing of Section 2.2 and Section 3.3. Alena Kopaničáková for her work on the UTOPIA solver modules and the integration of UTOPIA with the MOOSE library, and her contribution with the phase-phield example in Chapter 5. Dr. Maria Giuseppina Chiara Nestola for here help in integrating the parallel transfer algorithm of MOONOLITH with LIBMESH and MOOSE. Prof. Dr. Panayot Vassileski for the mentorship during my work at the Lawrence Livermore National Laboratory which led to the integration of MOONOLITH within their in-house software MFEM.

This work and the development of the related software libraries is partly supported by the Swiss National Science Foundation (http://www.snf.ch) under projects "Geometry-Aware FEM in Computational Mechanics" (No.:156178), "ExaSolvers – Extreme scale solvers for coupled systems" (No.:145271), "Parallel multilevel solvers for coupled interface problems" (No.:146167), "Large-scale simulation of pneumatic and hydraulic fracture with a phase-field approach" (No.:154090), and by the SCCER-FURIES program (http://sccer-furies.epfl.ch).

Contents

Chapter 1

Introduction

The finite element method [17; 44; 108] is a well established and known technique for the solution of partial differential equations. A great effort is invested in the development of software libraries implementing finite element assembly procedures and related solution algorithms. In fact, the development of such software libraries has several challenges.

The first challenge is dealing with complex mathematical models and simulating multiple physical phenomena simultaneously. This typically involves solving coupled systems of differential equations which might even require different discretizations (*e.g.,* molecular dynamics). The complexity of solving this problems rises when we introduce complex geometries having non-trivial interactions with each other, for instance contact between solids.

The second challenge is taking advantage of the concurrency within computational problems and taking advantage of the hardware resources available in modern super-computers. A significant amount of effort is invested in the development of parallel codes, and in new numerical methods/algorithms for optimally exploiting the available parallelism. As a consequence, scientific software becomes ever more complex and hard to reuse, re-purpose, maintain and extend.

The third challenge is the handling of geometric descriptions and the accuracy of their discrete representations. Embodying accurate representations with optimal and completely automatic black-box usage of state-of-the-art solvers is a non-trivial task.

The last challenge is modularity and usability of scientific software libraries. A reusable implementation of very complex algorithms is an important factor. The costs and effort of developing even only one such functionality might be significant. Hence, proper use of software design patterns and abstractions is

relevant for users at any level of involvement in the development of scientific codes. For instance, for solving a PDE with standard methods, users need only a minimal set of abstractions without having to deal with low level implementation details. Users that are researching new methods may however require access to specialized lower level abstractions. This aspects are strictly related to the issue of usability. It is often the case that scientific software imposes high barriers to entry for newcomers or inexperienced users. The presence of such high barriers is translated to poor productivity for new library adopters. For circumventing these challenges, a current trend in scientific software development is to strive for higher level abstractions.

This thesis is an attempt to contributing in dealing with the aforementioned challenges by covering three topics.

Parallel transfer of discrete fields for arbitrarily distributed unstructured finite element meshes

We present and investigate a new and *completely parallel* approach for the transfer of discrete fields between non-matching volume or surface meshes, arbitrarily distributed among different processors. No a priori information on the relation between the different meshes is required. Our inherently parallel approach is general in the sense that it can deal with both classical interpolation and variational transfer operators, *e.g.,* the L^2-projection and the pseudo-L^2-projection. It includes a parallel search strategy, output dependent load balancing, and the computation of element intersections, as well as the parallel assembling of the algebraic representation of the respective transfer operator. We describe our algorithmic framework and its implementation in the library MOONOLITH. Furthermore, we investigate the efficiency and parallel scalability of our new approach using different examples in 3D. This includes the computation of a volume transfer operator between 2 meshes with 2 billion elements in total and the computation of a surface transfer operator between 14 different meshes with 5.9 billion elements in total. The experiments have been performed with up to 12 288 cores.

Parametric finite elements with bijective mappings

We present a novel approach which combines parametric finite elements with smooth bijective mappings which allows to decouple the choice of approximation spaces from the geometric shape. Our approach allows to represent arbitrarily complex geometries on coarse meshes with curved edges, regardless of

the domain boundary complexity. The main idea is to use a bijective mapping for automatically warping the volume of a simple parameterization domain to the complex computational domain, thus creating a curved mesh of the latter. The numerical examples confirm that our method has lower approximation error than the standard finite element method, because we are able to solve the problem directly on the exact shape of domain without having to approximate it. In other words our method allows solving the model problem on the exact geometry with the freedom of choosing the discretization independently. This freedom enables to employ state-of-the-art solution strategies such as the multigrid method. Our discretization allows to automatically generate the meshes of a multigrid hierarchy just by refining a coarse mesh in the parameterization domain. This contribution is the result of a joint project and work with Teseo Schneider and Kai Hormann.

Utopia: a C++ embedded domain specific language for scientific computing

We present UTOPIA, a C++ embedded domain specific language designed for parallel non-linear solution strategies and finite element analysis. The rise of new computing hardware and the continuous development of numerical methods and programming technologies/languages/paradigms are drivers for changes in scientific-computing software libraries. However, such changes affect both the computing libraries and their dependencies, inducing unwanted modifications to high-level code. For avoiding these unwanted modifications, state-of-the-art software mainly relies on high-level programming interfaces or scripting languages. UTOPIA combines advantages of high-level programming interfaces with the advantages of scripting languages. On the one hand, it allows using high-level abstractions while providing access to the native low-level data-structures. On the other hand, it facilitates expressing complex numerical procedures by means of few lines of code. This is achieved by separating the model from the computation, thus allowing to keep the implementation details hidden from the code of applications such as non-linear solution algorithms and finite element assembly. We achieve this separation by using C++ meta-programming and particular evaluation strategies which allow mapping an abstract representation of the computation to the actual code computing the result. The linear algebra and finite element assembly codes snippets provides examples of the expressiveness of UTOPIA.

1.1 Thesis structure

In Chapter 2 we introduce the related work which includes mortar projection methods, parametric finite elements, and state of the art scientific software libraries. In Chapter 3 we described in detail a novel parallel algorithm for the transfer of discrete fields between arbitrarily distributed unstructured finite element meshes. In Chapter 4 we introduce a novel discretization based on parametric finite elements. In Chapter 5 we showcase the UTOPIA domain specific language and software library. In Chapter 6 we illustrate the performance studies of our parallel transfer algorithm and numerical experiments of our parametric finite element discretization. In Chapter 7 we briefly discuss general aspects of this thesis and its contributions.

Chapter 2

Geometry based techniques and abstraction tools in scientific software

In this chapter we introduce the related work. We describe the existing mathematical methods for exchanging information between finite element spaces (Section 2.1 and Section 2.2) and we provide a detailed introduction of the procedures (Section 2.3) and the geometric tools (Section 2.4) necessary to implement such methods. We briefly introduce existing methods for working with different geometric representations, and our method of choice for creating volume parameterizations (Section 2.5). We provide an overview of available open-source finite element software libraries from a software design/development perspective (Section 2.6).

2.1 Weak transfer between discrete spaces

The ever increasing computational power of modern super-computers allows, nowadays, for the numerical simulation of complex and coupled large scale problems, as arising from contact or fracture mechanics, fluid-structure interaction, computational geo-science, computational medicine, or, more general, multi-physics and multi-scale problems. Common to all these coupled and complex problems is the need for the transfer of data or information between the different models, meshes, or approximation spaces. The transfer of discrete fields as stresses, pressure, displacements, or velocities might be required along surfaces, *e.g.,* in the case of contact mechanics or fluid structure interaction, or within volumes, *e.g.,* in the case of transient simulations or multi-scale simulations. Ad-

ditionally, the transfer of discrete fields might also play an important role on the level of the discretization, *e.g.,* within non-conforming domain decomposition or mortar methods for the transfer along surfaces, or on the level of the solution method, *e.g.,* within multigrid or multi-level methods for the transfer between different volume meshes.

Clearly, the way transfer operators are constructed affects the quality of the used methods in terms of convergence, accuracy, and efficiency [53]. Thus, besides classical interpolation, more recently transfer operators based on variational approaches, such as (pseudo-)L^2-projections, have been developed. Here, in particular the mortar method [13] has to be mentioned, which has given rise to a huge number of new algorithmic developments during the last decades.

Despite these advances, deploying these approaches in a parallel high performance computing environment, the actual computation of such a volume or surface transfer operator turns out to be far from trivial. Different unstructured meshes might be arbitrarily distributed in a possibly unrelated manner, leading to many possible data distribution scenarios, which have to be handled in a transparent and efficient way. Additionally, the issues of scalability, usability and flexibility arise.

In our discussion we consider the general parallel case where we do not have any prior assumption on the spatial and memory location of the geometric objects. We focus our attention on the transfer of information of functional quantities from one mesh — or approximation space connected to a mesh — to another. Note that the actual choice of the approximation space may arise from finite elements, finite volumes or spectral methods. Here, we mainly consider the finite element method, as it is well known for dealing efficiently with complex unstructured geometries. For a more specific scenario we refer to [79], where the authors describe a technique for the parallel coupling between finite elements and molecular dynamics in a multi-scale method using a variational scale transfer.

In a parallel environment, the main challenges are identifying and handling relationships between geometric objects of interest based on spatial information, the used discretization and the application requirements. Given that a high degree of flexibility and generality is sought, the technical ingredients necessary for the realization of such strategies originates from different disciplines such as applied mathematics, geometric algorithms, software design, and high-performance computing.

There is a large number of different applications that might profit from a scalable parallel information transfer as presented herein:

- *Complex parallel multi-physics problems*. The handling of multiple types of

non-conformities in complex simulation scenarios with multiple geometries is usually done *ad-hoc*. A completely automated online strategy might allow more complex transient scenarios [63]. A common scenario, illustrated in the diagram in Figure 2.1(a), would be fluid-structure interaction and structure-structure interaction, where the fluid mesh is unstructured [64]. See [30; 94] for a comprehensive review of coupling methods in the context of fluid-structure interaction.

- *Coupling of distributed meshes* in non-conforming overlapping domain decomposition methods such as additive Schwarz [109], as illustrated by the diagram in Figure 2.1(b).

- *Handling of non-penetration conditions in parallel contact problem simulations* [138]. The contact surfaces between bodies are not always known a priori and they are in general geometrically non-conforming.

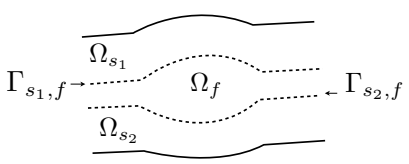
- *Parallel remeshing in transient simulations*, such as large deformations in computational mechanics. Local remeshing without having to ensure conformity at the subdomain interface allows for complete parallelization.

- *Handling of distributed multigrid hierarchies*. The coarse meshes can be selected without the restriction of requiring the same shape of the geometry or nested elements. The freedom to handle the various levels of refinement in a completely arbitrary way makes it possible to easily provide better balanced computations. For instance, as shown in Figure 2.1(c), cases where the hierarchy is generated by refining a coarse mesh into finer levels without balancing the computational load might lead to bad scaling. The approach we present in Chapter 3 allows constructing prolongation/restriction operators without requiring the additional programming of complex parallel code.

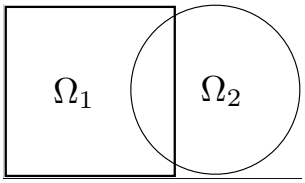
- *Multi-scale simulations, e.g.,* the coupling of molecular dynamics and finite elements as in [78; 139].

Additionally, numerical non-linear solution method can profit from non-conforming domain decomposition strategies such as in [54]. Extensive coverage of related matters, such as Galerkin projection methods, and intersection reporting can be found in [45; 46].

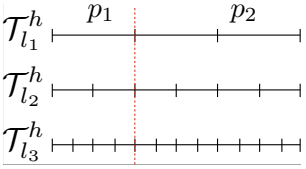
Let us comment on already existing approaches and their respective implementations. The question of variational information transfer has been addressed



(a) Fluid-structure interaction, where Ω_f is the fluid domain, and for $i \in \{1,2\}$, we have Ω_{s_i} and $\Gamma_{s_i,f}$ which are respectively structure domains, and fluid-structure interfaces.



(b) Overlapping decomposition, with two domains Ω_1 and Ω_2.



(c) 1D Multigrid hierarchy, with unbalanced load at the fine levels, where p_1 and p_2 are processes, and $\mathcal{T}_{l_i}$ is the mesh at level i.

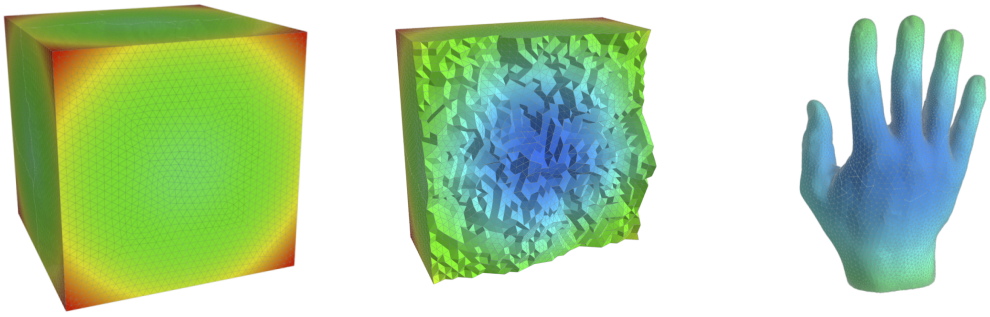
Figure 2.1. Simple example scenarios where our parallel approach can be applied.

in different numerical software, and software packages, such as DUNE, MOERTEL (TRILINOS package [59]), FENICS [89] project, OPENFOAM [135], and commercial software such as MPCCI [72], and COMSOL [86]. An abstract programming interface within the DUNE software for geometric coupling of finite element meshes is presented in [9]. The authors also bring to our attention the central problem of finding the geometric correspondences between meshes, and how in general it is solved by *ad-hoc* software solutions, with little chance of code reuse, *cf.* [9].

The next three sections provide a detailed introduction to the necessary tools that are the foundations of our parallel approach presented in Chapter 3. In Section 2.2, we summarize the main ideas of variational transfer. In Section 2.3, we illustrate how to assemble the local element-wise contributions to the resulting transfer operator both for volume transfer and contact problems. The assembly of such transfer operators require the computation of intersections between meshes. Thus, in Section 2.4, we introduce the most commonly adopted acceleration data-structures and algorithms for intersection detection.

2.2 Formulation

In the context of non-conforming domain decomposition methods, approaches using (pseudo-)L^2-projections, such as mortar methods [13; 137; 82] and their extensions for contact problems [34] and the literature cited therein, provide highly flexible ways for coupling possibly different discretizations across non-matching meshes. In our presentation, we focus on transferring discrete fields between finite element spaces associated with different unstructured meshes. We



(a) Source mesh. (b) Source mesh cut. (c) Target mesh.

Figure 2.2. Example of volume information transfer between different meshes. A given finite-element function on a cube (a) and (b), is transferred to a more complex geometry, *i.e.,* a hand (c).

note, however, that the techniques described herein are rather general and can be also applied to other types of discretizations, such as finite volume or spectral methods. Similar efforts for similar purpose include the Arlequin method [12].

2.2.1 Mortar projection

We start our discussion with a short introduction of the mortar projection, which will be used in our numerical experiments. For a comparison of different projection operators and their quantitative properties we refer the interested reader to [36].

For a (bounded) domain $\Omega \subset \mathbb{R}^d$ with Lipschitz boundary, let $L^2(\Omega)$ be, as usual, the Hilbert space of square integrable functions on Ω with inner product $(v,w)_{L^2(\Omega)} = \int_\Omega vw\,d\mathbf{x}$ and norm $\|\cdot\|_{L^2(\Omega)} = (\cdot,\cdot)^{1/2}_{L^2(\Omega)}$. Let $\Omega_m, \Omega_s \subset \mathbb{R}^d$ be bounded (Lipschitz) domains. Let the intersection $I = \Omega_m \cap \Omega_s$ of the two domains and the spaces $V = L^2(\Omega_m)$, $W = L^2(\Omega_s)$ be given.

We assume that Ω_m and Ω_s can be approximated, respectively, by the discrete domains Ω_m^h and Ω_s^h. Let the mesh $\mathcal{T}_k = \left\{E_k \subseteq \Omega_k^h \,|\, \bigcup \bar{E}_k = \bar{\Omega}_k^h\right\}$, with $k \in \{m,s\}$, be a finite set, where its elements E_k form a partition, hence if $E_k^1, E_k^2 \subseteq \Omega_k^h$ and $E_k^1 \neq E_k^2$ then $E_k^1 \cap E_k^2 = \emptyset$. For simplicity we consider $\mathcal{T}_k$, $k \in \{m,s\}$ to be conforming, though our approach is also applicable to the non-conforming case.

We denote the associated finite element spaces by $V_h = V_h(\mathcal{T}_m)$ and $W_h = W_h(\mathcal{T}_s)$. For non-matching meshes $\mathcal{T}_m$ and $\mathcal{T}_s$, also the approximation spaces V_h and W_h differ. We define the intersection of the two discrete domains as $I_h = \Omega_m^h \cap \Omega_s^h$, and assume that $I_h \neq \emptyset$. Furthermore, with $\mathcal{N}_m$ and $\mathcal{N}_s$ we denote the respective set of nodes of the meshes.

The case $\Omega_s^h \subseteq \Omega_m^h$

For simplicity, we now assume $\Omega_s^h \subseteq \Omega_m^h$. For this case, the projection has been shown to be stable [137]. We consider the case $\Omega_s^h \not\subseteq \Omega_m^h$ in Section 2.2.1. For the definition of the projection operator, we also need to define a suitable discrete space of Lagrange multipliers M_h. We here set $M_h = M_h(\mathcal{T}_s)$, *i.e.,* M_h is a discrete space based on the same mesh as W_h. The association of M_h with either $\mathcal{T}_m$ or $\mathcal{T}_s$ is arbitrary but fixed. Following the naming convention in the literature on mortar methods, the space associated with M_h, that is W_h, is often referred to as slave, or non-mortar, and the other one, that is V_h, as master, or mortar. The mortar projection maps a function from the mortar space, *i.e.,* V_h in our case, to the non-mortar space, *i.e.,* W_h.

Now we proceed to the definition of the projection operator $P : V_h \to W_h$. For a function $v_h \in V_h$ we want to find $w_h = P(v_h) \in W_h$, such that

$$(P(v_h), \mu_h)_{L^2(I_h)} = (v_h, \mu_h)_{L^2(I_h)} \quad \forall \mu_h \in M_h. \tag{2.1}$$

Reformulating Equation (2.1), *cf.* [13], we get the "weak equality" condition

$$\int_{I_h} (v_h - P(v_h))\mu_h \, d\mathbf{x} = \int_{I_h} (v_h - w_h)\mu_h \, d\mathbf{x} = 0 \quad \forall \mu_h \in M_h. \tag{2.2}$$

Let $\{\phi_i\}_{i\in J_v}$ be a basis of V_h, $\{\theta_j\}_{j\in J_w}$ of W_h, and $\{\psi_k\}_{k\in J_\mu}$ of M_h, where J_v, J_w, and $J_\mu \subset \mathbb{N}$ are index sets. Now writing the functions $v_h \in V_h$ and $w_h \in W_h$ in terms of the respective bases, we get $v_h = \sum_{i\in J^V} v_i \phi_i$, and $w_h = \sum_{j\in J^W} w_j \theta_j$, where $\{v_i\}_{i\in J_v}$ and $\{w_j\}_{j\in J_w}$ are real coefficients. This allows us to write the point-wise contributions to Equation (2.2) as

$$\sum_{i\in J_v} v_i \int_{I_h} \phi_i \psi_k \, d\mathbf{x} = \sum_{j\in J_w} w_j \int_{I_h} \theta_j \psi_k \, d\mathbf{x} \qquad \text{for } k \in J_\mu. \tag{2.3}$$

We rewrite Equation (2.3) as a matrix equation using the matrices $\mathbf{B}$ and $\mathbf{D}$ with respective entries $b_{k,i} = \int_{I_h} \phi_i \psi_k \, d\mathbf{x}$, and $d_{k,j} = \int_{I_h} \theta_j \psi_k \, d\mathbf{x}$, $i \in J_v$, $j \in J_w$, $k \in J_\mu$:

$$\mathbf{Bv} = \mathbf{Dw}. \tag{2.4}$$

Here, $\mathbf{v}$ and $\mathbf{w}$ are vectors of coefficients with respective entries v_i and w_j. From now on assume that the matrix $\mathbf{D}$ is square, that is $|J_w| = \left|J_\mu\right|$ and thus W_h and M_h have the same dimension. Additionally, we assume that $\mathbf{D}$ is invertible and thus we define the algebraic representation of the discrete (mortar) projection operator as $\mathbf{T} = \mathbf{D}^{-1}\mathbf{B}$ and rewrite Equation (2.4) as

$$\mathbf{w} = \mathbf{D}^{-1}\mathbf{Bv} = \mathbf{Tv}.$$

Depending on the choice of M_h, we obtain different transfer operators $\mathbf{T}$. For instance, using what is known as the dual basis for M_h, the matrix $\mathbf{D}$ becomes diagonal (or possibly block-diagonal for systems of equations). For details on this choice of M_h see Section 2.2.2.

The case $\Omega_s^h \not\subseteq \Omega_m^h$

For this case we do not provide stability guarantee on the projections. Due to $\Omega_s^h \not\subseteq \Omega_m^h$, we need to consider the extension to $\Omega_m^h \cup \Omega_s^h$ of the functions $v_h \in V_h$ by means of an extension operator. For Lipschitz domains, the existence of a continuous extension operator can be guaranteed [136, Theorem 5.3, Page 95]. In practice, different extension operators could be chosen, for example extension by zero, harmonic extension, or constant in the direction of the outer surface normal [75]. Eventually, this choice depends on the application.

Let $J_v^I = \{i \in J_v | \operatorname{supp}(\phi_i) \cap I_h \neq \emptyset\}$, $J_w^I = \left\{j \in J_w | \operatorname{supp}\left(\theta_j\right) \cap I_h \neq \emptyset\right\}$, and $J_\mu^I = \left\{k \in J_\mu | \operatorname{supp}(\psi_k) \cap I_h \neq \emptyset\right\}$ be the index sets of the basis functions of V_h, W_h, and M_h, respectively, with support in the intersection region I_h. By restricting the spaces V_h and W_h to I_h, we have the following new spaces

$$X_h = V_h|_{I_h} = \operatorname*{span}_{i \in J_v^I} \left\{\phi_i \cdot \chi_{I_h}\right\}, \quad Y_h = W_h|_{I_h} = \operatorname*{span}_{j \in J_w^I} \left\{\theta_j \cdot \chi_{I_h}\right\},$$

where χ_{I_h} is the characteristic function on I_h defined as

$$\chi_{I_h}(x) = \begin{cases} 1 \text{ if } x \in I_h, \\ 0 \text{ else.} \end{cases}$$

In order to adapt the definition of the projection operator to this case, we also define a modified version $\tilde{M}_h = \operatorname{span}_{k \in J_\mu^I} \left\{\tilde{\psi}_k\right\}$ of the multiplier space $M_h = \operatorname{span}_{k \in J_\mu} \{\psi_k\}$, where

$$\tilde{\psi}_k = \begin{cases} \psi_k|_{I_h} : \text{if } \operatorname{supp}(\psi_k) \subseteq I_h \\ \gamma_k : \text{if } \operatorname{supp}(\psi_k) \not\subseteq I_h \end{cases} \qquad \forall k \in J_\mu^I. \tag{2.5}$$

Here γ_k is a function defined on the intersection I_h. The functions γ_k are not necessarily the restrictions $\psi_k|_{I_h}$ of ψ_k to the intersection region I_h, but their definition depends on the choice of the multiplier space M_h. As M_h so far is a generic space, we here do not define γ_k. For an example construction in the case of the pseudo-L^2-projection see Section 2.2.2.

We can now adapt the definition of the projection operator P to the case $\Omega_s^h \not\subseteq \Omega_m^h$. For a function $v_h \in X_h$ we hence want to find $w_h = P(v_h) \in Y_h$, such that

$$(P(v_h), \tilde{\mu}_h)_{L^2(I_h)} = (v_h, \tilde{\mu}_h)_{L^2(I_h)} \quad \forall \tilde{\mu}_h \in \tilde{M}_h.$$

We can then derive the discrete projection operator $\mathbf{T}$ as in Section 2.2.1 under the assumption that W_h and $\tilde{M}_h$ have the same dimension. As a final remark, we note that with this definition of the spaces X_h and Y_h, the projected function $w_h = P(v_h) \in Y_h$ is by definition zero in $\Omega_s^h \setminus I_h$. Other extensions to $\Omega_s^h \setminus I_h$ are possible.

2.2.2 L^2-projection and pseudo-L^2-projection

In the preceding definition of the projection operator $\mathbf{T}$, we are still free to choose the multiplier space M_h. Different choices of M_h will lead to different projection operators. Setting for example $M_h = W_h$, $\mathbf{T}$ is the discrete representation of the L^2-projection. In this case, even though the mass matrix $\mathbf{D}$ is typically well-conditioned, the evaluation of $\mathbf{T} = \mathbf{D}^{-1}\mathbf{B}$ might become computationally expensive, or not convenient. It might be expensive because the inverse of $\mathbf{D}$ is dense. Hence, instead of storing $\mathbf{T}$, keeping $\mathbf{D}$ and $\mathbf{B}$ as separate matrices might be a better solution. However, this implies that each time we apply the transfer operator we solve a linear system. This is less convenient than storing only one matrix that can be applied directly.

We therefore consider mainly the case of choosing *dual basis* functions as a basis for M_h, as presented originally in [137]. In this case, the multiplier space M_h is spanned by a set of functions which are biorthogonal to the basis functions of W_h with respect to the L^2-inner product. This makes the matrix $\mathbf{D}$ diagonal, and computing its inverse cheap. In practice the matrix $\mathbf{D}$ is a lumped mass-matrix.

Since the vector space W_h is finite-dimensional, the dual basis exists, and the dimension of the dual space is the same as the one of the original space. In general, the dual basis functions ψ_k, $k \in J_\mu = J_w$, might have global support. Under certain assumptions on the space W_h, they can however be constructed elementwise in such a way that

$$\operatorname{supp}(\psi_k) \subseteq \operatorname{supp}(\theta_k) =: \omega_k \quad \forall k \in J_w \tag{2.6}$$

holds, *i.e.,* that their support is restricted to one finite element patch ω_k. This is for example possible assuming that W_h is the standard degree one finite element space and $\{\theta_j\}_{j \in J_w}$ is the standard basis [32].

In the case $\Omega_s^h \subseteq \Omega_m^h$, we choose the multiplier space as the discontinuous test space

$$M_h = \operatorname{span}\{\psi_k : I_h \to \mathbb{R} | \, k \in J_w, \operatorname{supp}(\psi_k) \subseteq \omega_k\} \not\subseteq C^0(I_h), \tag{2.7}$$

where the functions ψ_k satisfy the following biorthogonality condition:

$$(\psi_k, \theta_j)_{L^2(I_h)} = \delta_{j,k} (\theta_j, \mathbf{1})_{L^2(I_h)} \quad \forall \, j,k \in J_w. \tag{2.8}$$

As described in [32; 47], a basis $\{\psi_k\}_{k \in J_w}$ fulfilling (2.7) and (2.8) can be constructed in a straightforward way, using only computations on single elements. Let $E \in \mathscr{T}_s$ be one element in the mesh of the finite element space W_h. Let $M_E = \left(m_{pq}\right)$ be an element mass matrix, and $D_E = \left(d_{pq}\right)$ be an element diagonal matrix defined by

$$m_{pq} = \left(\theta_p, \theta_q\right)_{L^2(E)}, \quad d_{pq} = \delta_{pq} \left(\theta_p, \mathbf{1}\right)_{L^2(E)} \quad \forall p,q \in \mathscr{N}_s \cap \overline{E},$$

respectively, where $\mathscr{N}_s$ are the nodes of $\mathscr{T}_s$, and $\overline{E}$ is the closure of the element E of mesh $\mathscr{T}_s$. As M_E is symmetric positive definite and thus invertible, for $p \in \mathscr{N}_s \cap \overline{E}$ we can define functions $\psi_{p,E}$ by

$$\psi_{p,E}(\mathbf{x}) := \begin{cases} \sum_{r \in \mathscr{N}_s \cap \overline{E}} \left(D_E M_E^{-1}\right)_{pr} \theta_r(\mathbf{x}) & \text{if } \mathbf{x} \in E, \\ 0 & \text{else.} \end{cases} \tag{2.9}$$

Then we can define the dual basis fulfilling (2.7) and (2.8) by

$$\psi_p = \sum_{E \in \mathscr{T}_s : p \in \overline{E}} \psi_{p,E} \quad \forall p \in \mathscr{N}_s. \tag{2.10}$$

We furthermore note that in the case of affine elements, due to the scaling with $(\theta_j, \mathbf{1})_{L^2(I_h)}$ on the right-hand side of Equation (2.8), the coefficients in Equation (2.9) do not depend on the element E or the node p [32]. Thus it is sufficient to compute them only once on the reference element. Furthermore, in this case, the dual basis function ψ_k is continuous on the patch ω_k, that is $\psi_k|_{\omega_k} \in C^0(\omega_k)$.

In the case where $\Omega_s^h \not\subseteq \Omega_m^h$, and $I_h = \Omega_s^h \cap \Omega_m^h \neq \emptyset$, we provide an example for a modified multiplier space. We would like to stress that our framework is general in the sense that multiplier and approximation spaces can be freely prescribed by the user. In our example, let the discontinuous test space be

$$\tilde{M}_h = \operatorname{span}\{\tilde{\psi}_k : I_h \to \mathbb{R} | \, k \in J_w^I, \operatorname{supp}\left(\tilde{\psi}_k\right) \subseteq \tilde{\omega}_k\} \not\subseteq C^0(I_h), \tag{2.11}$$

where $\tilde{\omega}_k$ is the support of the k-th basis function of Y_h, and the functions $\tilde{\psi}_k$ with support in $\tilde{\omega}_k$ satisfy the following biorthogonality condition:

$$(\tilde{\psi}_k, \theta_j)_{L^2(I_h)} = \delta_{p,q} (\theta_j, \mathbf{1})_{L^2(I_h)} \quad \forall \, j,k \in J_w^I. \tag{2.12}$$

Hence $\{\tilde{\psi}_k\}$ is the dual basis with respect to the basis $\{\theta_k \cdot \chi_{I_h}\}$ of Y_h.

As in the previous case, we can construct the dual basis elementwise by slightly modifying the above procedure. More precisely, we restrict all indices to the smaller index set J_w^I, and replace θ_k by $\theta_k \cdot \chi_{I_h}$ for all k. This implicitly defines the functions γ_k in Equation (2.5).

In this case, even for affine elements, for an element E that is not completely contained in the intersection, *i.e.,* $E \not\subseteq I_h$ and $E \cap I_h \neq \emptyset$, the coefficients in the modified Equation (2.9) do depend on the element and on the node. Thus the local matrices D_E and M_E need to be computed and M_E needs to be inverted on every such element separately. Moreover, this implies that the function $\tilde{\psi}_k$ is in general not continuous on its support. If $E \cap I_h$ is small, the jump in the function $\tilde{\psi}_k$ might become large, leading to instabilities in the method. This problem can be handled by considering intersections with really small volume as empty. Numerically speaking, we consider the intersections $\mathrm{supp}(.) \cap I_h$ to be empty, if their volumes are smaller than a small numerical constant. We emphasize that this is an *ad-hoc* solution, which has turned out to work well in practice, which does no affect the overall approach.

The pseudo-L^2-projection is a projection, and it also guarantees an efficient evaluation of the transfer operator $\mathbf{T}$. In fact, using dual basis functions, $\mathbf{T}$ can be evaluated easily, as $\mathbf{D}$ becomes diagonal (or block-diagonal in the case of systems). Thus, the usage of dual basis functions corresponds to replacing the standard L^2-projection by a pseudo-L^2-projection, which allows for a more efficient assembly and application of $\mathbf{T}$.

As investigated numerically in the study performed in [36], the pseudo-L^2-projection is close to the L^2-projection in terms of the operator norm. The pseudo-L^2-projection is also proven to be H^1-stable and has the L^2-approximation property for all shape-regular families of meshes (see [137; 32] for more details). All of the numerical experiments presented in this thesis employ this operator.

2.2.3 Relation to the application scenarios

All the application scenarios we mentioned can be categorized either as volume projection or as surface projection. Here we provide a link to the mathematical objects we presented in Section 2.2.

Volume projections

Information transfer between volumes (*i.e.,* volume projections) can be directly related to the operators introduced above, hence allowing us to transfer informa-

tion between finite element discretizations from one volume to another volume, as illustrated in the example in Figure 2.2. In fact, it is sufficient to consider $\mathcal{T}_m$ and $\mathcal{T}_s$ as volume meshes in N dimensions.

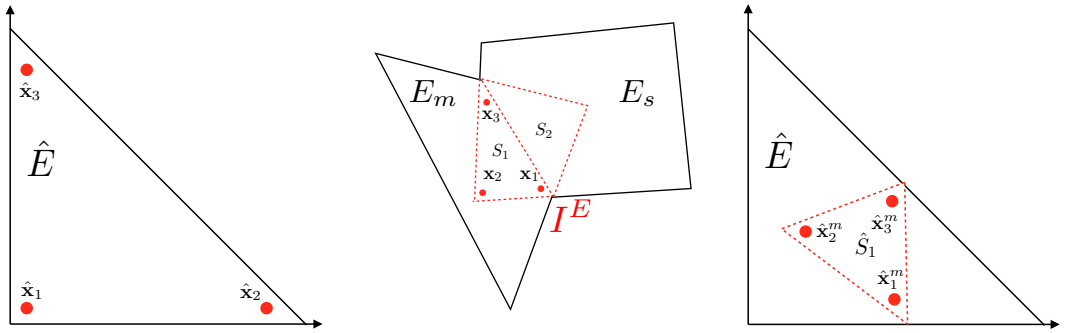
Surface projections

Information transfer between non-matching surface meshes (*i.e.,* surface projections) shows up in many different applications. These might be coupled problems, such as, *e.g.,*, fluid structure interaction or contact problems. For fluid-structure interaction, two different meshes are used for the fluid and solid. In this case, usually surface forces originating from the fluid have to be transferred to the solid and the velocities of the solid have to be transferred to the fluid domain.

For contact and tying problems, boundary stresses and boundary displacements have to be transferred between the two interacting bodies. We refer to [33; 106; 105; 34] and the literature cited therein concerning different approaches for the treatment of surface projections in the framework of contact problems with non-conforming contact interfaces. An alternative method for contact and tying problems is typically the NTS (node-to-segment) method. However, the NTS method exhibts deficiencies such as failure to pass the patch test and oscillatory stress response which are not present in mortar methods [60; 61].

What is common to both fluid structure interaction and contact problems, is that the two surface meshes under consideration in general will also be non-matching with respect to their position in space. For instance, in contact problems we have surface meshes which are in general non-matching on the predicted area of contact. Thus, it will also be necessary to project the function values in "physical space" between the two surfaces. Usually, this is done by means of a normal projection. However, the way this normal projection is realized and the way it is incorporated into the quadrature routines needed for assembling the matrices **B**, **D** has strong influence on the quality of the resulting projection operator **T**, *cf.* [33; 106; 105; 34].

Thus, surface transfers are not simply volume transfers in $2D$, but, additionally involve the careful construction of a discrete (normal) projection.



(a) Quadrature points $\hat{\mathbf{x}}_i$ on reference element $\hat{E}$.

(b) Quadrature points $\mathbf{x}_i = G_{\hat{E}\to S_i}(\hat{\mathbf{x}}_i)$ mapped to the simplex $S_i \in \mathscr{T}_{I^E}$.

(c) Quadrature points $\hat{\mathbf{x}}_i^m = G^{-1}_{\hat{E}\to E_m}(\mathbf{x}_i)$ in the reference element $\hat{E}$.

Figure 2.3. Overview of the quadrature data for the assembly. The quadrature points mapped to the simplex S_i are transformed to the reference element for evaluating the basis functions.

2.3 Procedure for the assembly of the coupling operators

In this section, we describe in detail one example procedure for the assembly of the matrices $\mathbf{B}$ and $\mathbf{D}$ defining the coupling operator $\mathbf{T} = \mathbf{D}^{-1}\mathbf{B}$ for the case of affine finite element discretizations associated to the two non-conforming meshes $\mathscr{T}_m$, $\mathscr{T}_s$. We choose $\mathscr{T}_m$ to be the master, and $\mathscr{T}_s$ to be the slave, where m stands for master, and s stands for slave. As before, the finite element spaces associated with these meshes are $V_h(\mathscr{T}_m)$, $W_h(\mathscr{T}_s)$, and the multiplier space is $M_h(\mathscr{T}_s)$.

The assembly is done in four main steps.

1. We determine all pairs of intersecting elements $\langle E_m, E_s\rangle$, $E_m \in \mathscr{T}_m$ and $E_s \in \mathscr{T}_s$. This can be done by means of tree-search algorithms and data-structures (quadtree, octree), or by means of advancing-front algorithms with linear complexity such as the one proposed in [49].

2. For each pair $\langle E_m, E_s\rangle$, we compute the intersection polytope $I^E = E_m \cap E_s$ of the two intersecting elements E_m and E_s and we mesh it (for our convenience, triangulate it in $2D$ and for surface projections, or tetrahedralize it in $3D$). Hence, we obtain the mesh

$$\mathscr{T}_{I^E} = \left\{S_i \subseteq I^E \mid \bigcup \bar{S}_i = \bar{I}^E \text{ and } S_i \text{ is a simplex}\right\}$$

where for $S_i, S_j \subseteq I^E$ if $S_i \neq S_j$ then $S_i \cap S_j = \emptyset$. The computation of the intersection polytope can be done by means of the Sutherland-Hodgman

clipping algorithm [130]. Note that the mesh $\mathscr{T}_{I^E}$ does necessarily has to be explicitly created, the next step can be performed by treating each simplex implicitly by only using the intersection polytope connectivity.

3. We generate the quadrature points for integrating in the intersection region I^E. This can be done by mapping points from quadrature rules defined on the reference simplex $\hat{E}$ to each simplex S_i.

4. We compute the local element-wise contributions by means of numerical quadrature and assemble the two matrices **B** and **D**.

We now focus exclusively on the details of the last two steps, that is on the assembly of the operators with respect to a given pair of elements $\langle E_m, E_s \rangle$ and their intersection I^E. We start by choosing a suitable quadrature formula (such as Gaußian quadrature [127]), with K points $\{\hat{\mathbf{x}}_k\}_{k=1}^K \subseteq \hat{E}$ and weights $\{\alpha_k\}_{k=1}^K$ with $\sum_{k=1}^K \alpha_k = 1$. An example quadrature formula is shown in Figure 2.3(a). Then, for each simplex S_i:

- We map the quadrature points $\{\hat{\mathbf{x}}_k\}_k \subset \hat{E}$ to S_i obtaining $\{\mathbf{x}_k\}_k \subset S_i$ as shown in Figure 2.3(b).

- We transform $\{\mathbf{x}_k\}_k \subset S_i \subseteq E_m \cap E_s$ to the reference element for both elements: $\hat{\mathbf{x}}_k^m = G^{-1}_{\hat{E}\to E_m}(\mathbf{x}_k)$ and $\hat{\mathbf{x}}_k^s = G^{-1}_{\hat{E}\to E_s}(\mathbf{x}_k)$, where $G_{\hat{E}\to E_i}$, $i \in \{m,s\}$, is the transformation from the reference element $\hat{E}$ to the element E_i as shown in Figure 2.3(c).

- We set weights

$$\alpha'_k := \alpha_k |\hat{E}| \left|\det(\nabla G_{\hat{E}\to E_s}(\hat{\mathbf{x}}_k^s))\right| |S_i| \,/\, |E_s|\,,$$

  where by $|X|$ for $X \subseteq \mathbb{R}^d$ we denote the volume of X.

- We compute and add the local contributions to the global coupling matrices as follows

$$b_{p,q} \leftarrow b_{p,q} + \sum_{k=1}^K \alpha'_k \hat{\psi}_p(\hat{\mathbf{x}}_k^s)\hat{\phi}_q(\hat{\mathbf{x}}_k^m),$$

$$d_{p,q} \leftarrow d_{p,q} + \sum_{k=1}^K \alpha'_k \hat{\psi}_p(\hat{\mathbf{x}}_k^s)\hat{\theta}_q(\hat{\mathbf{x}}_k^s),$$

  the matrix entries at p,q for **B** and **D** respectively, where $\hat{\psi}_p$, $\hat{\phi}_q$, $\hat{\theta}_q$ are basis function defined in the reference element. The respective global counterparts are $\phi_p \in V_h(E_m)$, $\theta_q \in W_h(E_s)$, and $\psi_q \in M_h(E_s)$ which is the Lagrange multipliers basis associated to E_s.

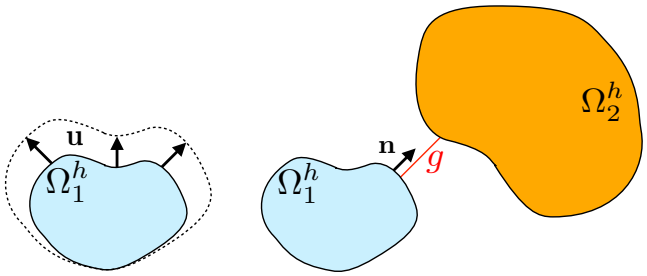


Figure 2.4. Displacement $\mathbf{u}$, surface normal vector $\mathbf{n}$ and gap g.

2.3.1 Assembly procedure for two-body contact problems

The use of mortar methods in contact simulations requires, not only a more involved selection of candidates, but also a more involved assembly procedure [34]. Let us consider a two-body contact problem, between two linear elastic bodies. The two bodies are conveniently denoted as $\Omega_m \subset \mathbb{R}^d$ and $\Omega_s \subset \mathbb{R}^d$, $\Omega_m \cap \Omega_s = \emptyset$. The displacement field $\boldsymbol{u}$, decomposed into $\boldsymbol{u}^m$ and $\boldsymbol{u}^s$, is given as the solution to the boundary value problem

$$\begin{aligned} -\operatorname{div}\boldsymbol{\sigma}(\boldsymbol{u}) &= \boldsymbol{f} \quad \text{in } \Omega \\ \boldsymbol{u} &= \boldsymbol{q} \quad \text{on } \Gamma^D \\ \boldsymbol{\sigma}(\boldsymbol{u})\boldsymbol{n} &= \boldsymbol{p} \quad \text{on } \Gamma^N, \end{aligned} \tag{2.13}$$

where $\boldsymbol{\sigma}$ is the stress tensor incorporating the material law, $\boldsymbol{n}$ is the outer surface normal, $\Omega = \Omega_m \cup \Omega_s$ and $\Gamma = \Gamma_s \cup \Gamma_m$, with $\Gamma_s \cap \Gamma_m = \emptyset$, represent the boundary of Ω. With Γ^D we denote the Dirichlet boundary, with Γ^N the Neumann boundary and with Γ^C the contact boundary, $(\Gamma^D \cap \Gamma^N) \cup (\Gamma^D \cap \Gamma^C) \cup (\Gamma^N \cap \Gamma^C) = \emptyset$. We cover linearized contact conditions which do not apply to more general nonlinear problems. Such conditions are constructed by considering a very specific set of contact directions defined by the normal field on Γ_s^C which leads to the following definition of gap function $g : \Gamma_s^C \to \mathbb{R}$, with

$$g(\boldsymbol{x}) = \min_{\boldsymbol{x}_m \in \Gamma_m^C} \boldsymbol{n}(\boldsymbol{x})^T(\boldsymbol{x}_m - \boldsymbol{x}),$$

and the following non-penetration condition, $\forall \boldsymbol{x} \in \Gamma_s^C$

$$\boldsymbol{n}(\boldsymbol{x})^T(\boldsymbol{u}^s(\boldsymbol{x}) - \boldsymbol{u}^m(\boldsymbol{y})) \leq g(\boldsymbol{x}), \tag{2.14}$$

where $\boldsymbol{y} = \operatorname{argmin}_{\boldsymbol{x}_m \in \Gamma_m^C} \boldsymbol{n}(\boldsymbol{x})^T(\boldsymbol{x}_m - \boldsymbol{x})$. We consider a frictionless contact problem, hence on Γ^C the tangential components of the stress are expected to be

equal to zero, and the normal component to be less or equal to zero. Figure 2.4 provides a visual representation of these quantities.

We discretize Ω_m and Ω_s with respective meshes $\mathscr{T}_m$ and $\mathscr{T}_s$. With $\boldsymbol{V}_h = V_h^d(\mathscr{T}_m)$, $\boldsymbol{W}_h = W_h^d(\mathscr{T}_s)$, and $\boldsymbol{M}_h = M_h^d(\mathscr{T}_s)$, where d corresponds to the spatial dimension, we denote tensor-product spaces. Let $\boldsymbol{u}_h^m \in \boldsymbol{V}_h$ and $\boldsymbol{u}_h^s \in \boldsymbol{W}_h$ represent the discete displacement fields in the master and slave mesh respectively.

The assembly procedure of the coupling matrices $\boldsymbol{B}$ and $\boldsymbol{D}$, the weighted gap block-vector $\boldsymbol{g}^M \in M_h^d(\mathscr{T}_s)$ and weighted normal block-vector $\boldsymbol{n}^M \in \boldsymbol{M}_h$ consists of the following steps:

1. we determine all the pairs of near surface elements $\langle E_m, E_s \rangle$, $E_m \in \mathscr{T}_m$ and $E_s \in \mathscr{T}_s$. We employ the same strategies mentioned in Section 2.3, *i.e.,* octrees and spatial-hashing, but we enlarge the bounding volumes of the surface elements by small amounts in both positive and negative normal directions.

2. Let E_s be a planar surface element with normal $\boldsymbol{n}$, which defines a projection plane. For simplicity we perform our computation in a $(d-1)$-dimensional setting. Hence, if $d = 3$, we compute the affine map $G(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{q}_1$, where $\boldsymbol{A} = [\boldsymbol{w}, \boldsymbol{v}, \boldsymbol{n}]$, where $\boldsymbol{w} = \boldsymbol{q}_2 - \boldsymbol{q}_1, \boldsymbol{v} = \boldsymbol{q}_3 - \boldsymbol{q}_1, \boldsymbol{n} = (\boldsymbol{w} \times \boldsymbol{v})/\|\boldsymbol{w} \times \boldsymbol{v}\|_2$, and $\boldsymbol{q}_i, i = 1, \ldots, n$ are the vertices of the element E_s. Note that $G^{-1}(E_s) = \hat{E} \subset \mathbb{R}^{d-1}$ is the reference element of the surface element E_s. For the sake of brevity, we denote the set $\{G(\boldsymbol{q})\}, \boldsymbol{q} \in Q$, where Q is a set of points, as $G(Q)$. We transform the master surface element and obtain obtaining $E_G^m = G^{-1}(E_m)$, from which we remove the last component from all its vertices and obtain the $(d-1)$-dimensional orthogonal projection $\hat{E}^m$.

   We find the intersection $\hat{I} = \hat{E} \cap \hat{E}^m$. If $\hat{I} = \emptyset$ we stop. Otherwise, for the slave side we compute $I_s = G(\hat{I})$. For the master side we compute the orthogonal projection of $\hat{I}$ onto the surface defined by E_G^m, the result is then transformed by G to world coordinates thus obtaining I_m.

3. Once we have I_s and I_m we compute the coupling operators $\boldsymbol{D}$ and $\boldsymbol{B}$ as in the procedure illustrated in Section 2.3 by following step 3 and 4.

4. We assemble the weighted direction vector and weighted gap vector as

follows:

$$\boldsymbol{n}_p^M \leftarrow \boldsymbol{n}_p^M + \sum_{k=1}^{K} \alpha_k' \hat{\boldsymbol{\mu}}_p(\hat{\mathbf{x}}_k^s) \cdot \mathbf{n}(\mathbf{x}_k^s)$$
$$\boldsymbol{g}_p^M \leftarrow \boldsymbol{g}_p^M + \sum_{k=1}^{K} \alpha_k' \hat{\boldsymbol{\mu}}_p(\hat{\mathbf{x}}_k^s) \cdot \boldsymbol{e}_1 g(\mathbf{x}_k^s),$$

where $\boldsymbol{e}_1 = [1,0,0]^T$, $\hat{\boldsymbol{\mu}}_p$ is the counterpart of $\boldsymbol{\mu}_p \in \boldsymbol{M}_h$ in the reference element, and $p \in J_C^s$, $J_C^s = \{j \in \mathcal{N}_s | \operatorname{supp}(\mu_j) \cap \Gamma_C \neq \emptyset\}$. The d-dimensional blocks of vector $\boldsymbol{n}^M$ are normalized after assembly.

Note that if $p \notin J_C^s$ we consider all the associated elements in $\boldsymbol{B}$, $\boldsymbol{D}$, $\boldsymbol{n}^M$ as zero. For the gap vector $\boldsymbol{g}^M$ we set the entries of $p \notin J_C^s$ to a suitable large number $\eta \in \mathbb{R}_{>0}$ and to $[\boldsymbol{g}_p^M \cdot \boldsymbol{e}_1, \eta, \eta]$ otherwise. We consider the indices in J_C^s to be contiguous so that we can visualize the results as

$$\boldsymbol{B} = \begin{bmatrix} \mathbf{0} & \boldsymbol{B}_C \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \boldsymbol{D} = \begin{bmatrix} \boldsymbol{D}_C & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \boldsymbol{n}_M = \begin{bmatrix} \boldsymbol{n}_C \\ \mathbf{0} \end{bmatrix}, \boldsymbol{g}_M = \begin{bmatrix} \boldsymbol{g}_C \\ \boldsymbol{\eta} \end{bmatrix}.$$

We compute the coupling operator $\boldsymbol{T} = \boldsymbol{D}^g \boldsymbol{B} + \mathbf{Id}$, where $\boldsymbol{D}^g$ is the generalized inverse of $\boldsymbol{D}$, the gap-vector coefficients $\boldsymbol{g} = \boldsymbol{D}^g \boldsymbol{g}^M$, and the block-vector $\boldsymbol{D}^g \boldsymbol{n}^M$ which is then normalized block-wise to obtain the block-vector of normals $\boldsymbol{n}$.

Since we have chosen to assign what we call normal component to the first coordinate of each block of the gap vector. In fact we assume, a normal-tangential coordinate system (frame of reference) spanned by the mutually orthogonal vectors $\boldsymbol{n}_p, \boldsymbol{t}_p^1, \boldsymbol{t}_p^2$ which are respectively the weighted surface normal an the respective tangential vectors at the node p.

Let us consider the following linear system of equations $\boldsymbol{Au} = \boldsymbol{f}$ arising from our contact problem (2.13). In order to work with the non-penetration condition we transform the systems of equation. We do this by means of the Householder transformation $\boldsymbol{O}_{pp} = \mathbf{Id} - 2\boldsymbol{w}\boldsymbol{w}^T, \boldsymbol{w} = (\boldsymbol{n}_p + \boldsymbol{e}_1)/\|\boldsymbol{n}_p + \boldsymbol{e}_1\|_2$, $\boldsymbol{O}_{pp} = \boldsymbol{O}_{pp}^T = \boldsymbol{O}_{pp}^{-1}$ for $p \in J_C^s$, and by $\boldsymbol{O}_{pp} = \mathbf{Id}$ otherwise. We finally write the algebraic formulation of (2.13) as

$$\boldsymbol{O}\boldsymbol{T}^T\boldsymbol{A}\boldsymbol{T}\boldsymbol{O}\tilde{\boldsymbol{u}} = \boldsymbol{O}\boldsymbol{T}^T\boldsymbol{f},$$
$$\tilde{\boldsymbol{u}} \leq \boldsymbol{g}$$

where $\tilde{\boldsymbol{u}} = \boldsymbol{O}\boldsymbol{T}^T\boldsymbol{u}$, which we solve by means of any method which handles inequality constraints, such as projected gauss-seidel, non-linear multigrid, or semi-smooth Newton methods [138; 107].

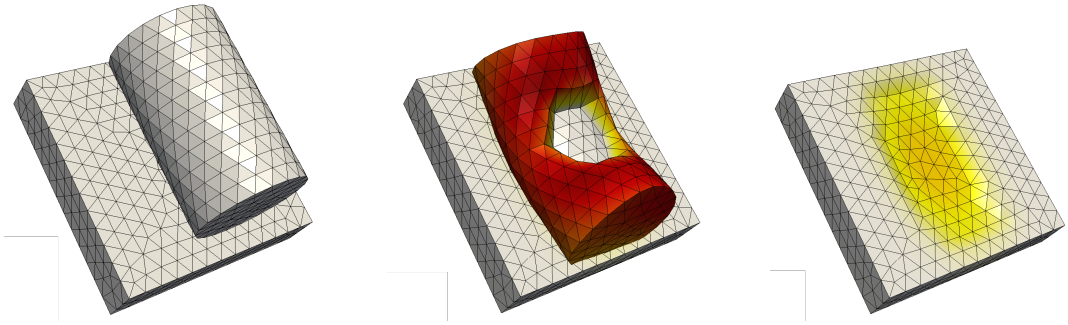


Figure 2.5. Contact simulation between a parallelepiped and a cylinder with automatically determined contact.

Automatic determination of contact patches

When using the mortar method in contact problems the role of adjacent surface elements has to be the same, *i.e.,* if an element is assigned the master role all its adjacent elements can not be assigned the slave role, and vice-versa for an element with the slave role.

When the slave and master roles are not provided a-priori by the user, they are determined automatically. An example of such situation is self-contact in transient scenarios. In such cases it is natural to consider the element describing different bodies as part of a unique mesh. A strategy for automatically handling this assignment problem is presented in [141]. We describe a more basic strategy which consists of three main steps.

The first step consists of rejecting pairs of element which are detrimental to the quality of the discretized non-penetration constraints. One criterion is to reject pairs of elements that have a common node. Another criterion is to reject pairs of elements for which $\cos\theta > \beta$, where $\cos\theta = \mathbf{n}_s^T \mathbf{n}_m$ is the cosine of the angle between the normals $\mathbf{n}_s$ and $\mathbf{n}_m$ defined on the slave and master surface elements respectively, and $\beta \in \mathbb{R}_{<0}$.

The second step consists of identifying which elements are suitable to be assigned the slave role. An element E_s is considered suitable if its area $|E_s|$is approximately equal to the total area of the geometric normal projection $\sum_{k=1}^{K} |I_s^k|$ (Section 2.3.1), where K is the number of geometric projections. This step gives us a weighted directed graph C with n vertices, which we call contact graph, where each vertex corresponds to an element, and each edge goes to a valid slave candidate E_i to a valid master candidate E_j . We consider the weight c_{ij} associated to the edge (i, j) of C to be the average gap function from the slave

Input Result

Figure 2.6. Contact simulation between multiple bodies with automatically determined contact.

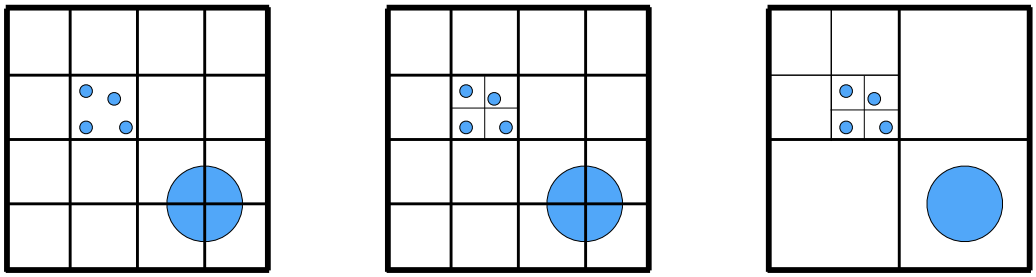
candidate to the master candidate. For a vertex i we define the set of vertices connected by its outgoing-edges as C_i.

The third and last step, is the actual master-slave role assignment. For prioritizing near surfaces, we first consider the vertices of the contact graph (*i.e.,* candidate slave elements) that have outgoing edges with small weights. In other words, we consider E_i before E_j if $\sum_{k \in C_i} c_{ik} < \sum_{k \in C_j} c_{jk}$. If an element E_i without role has has either adjacent master elements or is connected to a slave element through an edge of C we consider E_i to be ambiguous. We visit each candidate slave element E_i and we check if it has no role assigned. If E_i is ambiguous we skip it. We assign the slave role to E_i, then immediately visit its neighboring elements by traversing the adjacency graph defined by the mesh in a breadth-first manner until we encounter candidate slave elements without role and that are not ambiguous. We consider elements to be adjacent if they share a common side. Then, for each parir $c_{ij}, j \in C_i$ we set the master role to the element E_j and all its neighboring elements using the same breadth-first traversal strategy we previously described. Figure 2.5 and Figure 2.6 show examples of this strategy applied to a two-body and multiple-body contact problem respectively.

2.3.2 Non–affine elements and quadrature points

In Section 2.3 we described the assembly procedures that only account for elements with affine geometric maps. The main reason is that computing the intersection between non-affine (curved) elements is both non-trivial and computationally more expensive.

A possible solution is to discretize such elements into piecewise linear polygons (or polyhedra), intersect the polygonal approximations and generate the quadrature points from the resulting intersection as in Section 2.3. Note that the polygonal approximation might be non-convex, consequently the intersec-



Uniform grid. Hierarchical grid. Quadtree.

Figure 2.7. Space partitioning strategy examples for a five body data-set.

tion of two such polygons might be non-convex either. Hence, for meshing the intersection more elaborate meshing strategies [27; 123] are required.

In [34] the authors show how to deal with contact between warped surfaces by approximating the contact condition thorough the introduction of a common plane. An option is the best fitting plane of the slave surface element or the plane defined by the center of the element and its normal as in [105]. The geometric projection is performed on the convex hull of the corners of the elements. After the intersection is computed the back-projection is performed by solving a non-linear problem.

2.4 Space partitioning and ordering

A performance critical aspect for exploiting the information transfer methods described in Section 2.1 is intersection detection. Thus, in this section we provide an overview of many relevant intersection detection techniques.

2.4.1 Space-subdivision strategies and acceleration data–structures

Acceleration data-structures and algorithms are widely used in spatial problems, and there is a great amount of literature covering the topic [29; 41; 101; 131]. In this section, we introduce some of the most commonly adopted objects for intersection detection, such as bounding volumes, grids, and binary space-partitioning trees. A disadvantage of grids and trees is that you have to deal with the complication of objects intersecting multiple partitions. Sort and sweep methods [29] avoid this complication. However, the performance of such algorithms breaks down with respect to some common positional scenarios, hence they are completely neglected in this thesis.

2.4.2 Bounding volumes

A bounding volume is a closed volume completely containing a set of geometric objects. Testing a bounding volume for intersections has to be significantly cheaper than testing the contained objects. Commonly used bounding volumes are bounding-spheres and convex-hulls. In this thesis we cover exclusively the discrete oriented polytope (DOP) and the axis-aligned bounding-box (AABB). The k-DOP is a discrete oriented polytope described as the intersection of k half-spaces, see Figure 2.8. The AABB can be considered as a special case of the k-DOP where the half-space orientations are given by the canonical basis vectors. More specifically, a k-DOP $\mathscr{B}$ is defined as a set of k normal vectors $[\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_k]$ and signed distances from the origin of the respective cutting planes. We denote the minimum distances as $\boldsymbol{B}^m = [B_1^m, B_2^m, \ldots, B_k^m]$ and the maximum distances as $\boldsymbol{B}^M = [B_1^M, B_2^M, \ldots, B_k^M]$. The k-DOP of a set of points Q is computed as $B_i^m = \min_{\boldsymbol{q}\in Q} \boldsymbol{b}_i \cdot \boldsymbol{q}$ and $B_i^M = \max_{\boldsymbol{q}\in Q} \boldsymbol{b}_i \cdot \boldsymbol{q}$, $i = 1, \ldots, k$. For a pair of k-DOPs $\mathscr{A}$ and $\mathscr{B}$ if

$$\exists_{i=1,\ldots,k} \quad A_i^M < B_i^m \quad \vee \quad B_i^M < A_i^m, \tag{2.15}$$

is satisfied, then there is no intersection.

2.4.3 Spatial hashing

Spatial hashing data-structures, such as implicit grids, allow having constant computational time complexity spatial queries for several applications, such as 3D parameterized textures, 3D painting, collision and intersection detection. Here, we focus on the latter application. There are several strategies for performance reliable spatial hashing, for computations both on CPU and GPU, such as perfect hashing [84].

For uniformly or quasi-uniformly sized and distributed data, spatial hashing provides the fastest way of detecting intersections. The simplest spatial-hashing data-structure is a uniform implicit grid, which we refer to as hash-grid. We recall the definitions of k-DOP and AABB introduced in Section 2.4.2. The hash-grid is constructed by dividing each dimension k of the axis-align bounding-box $B = [\mathbf{B}^m, \mathbf{B}^M]$ in

$$n^k = \left\lfloor \left\lceil \frac{1}{2}\left(\frac{N}{2}\right)^{1/s} \right\rceil \frac{B_k^M - B_k^m}{\min\limits_{l=1,\ldots,s} (B_l^M - B_l^m)} \right\rfloor$$

intervals which creates $n = \prod_{k=1}^{s} n^k$ cells, where B_k^m, B_k^M denote the k-th components of the respective vectors, $\lfloor\cdot\rfloor$ is the floor operator, and $\lceil\cdot\rceil$ is the ceiling

operator. The hash function is of the form

$$h(\boldsymbol{x}) = \sum_{k=1}^{d} \left(\mathscr{I}_k(\boldsymbol{x}) \prod_{l=k+1}^{d} n^l \right),$$

where

$$\mathscr{I}_k(\boldsymbol{x}) = \left\lfloor (x^k - B_k^m)/(B_k^M - B_k^m) \cdot n^k \right\rfloor$$

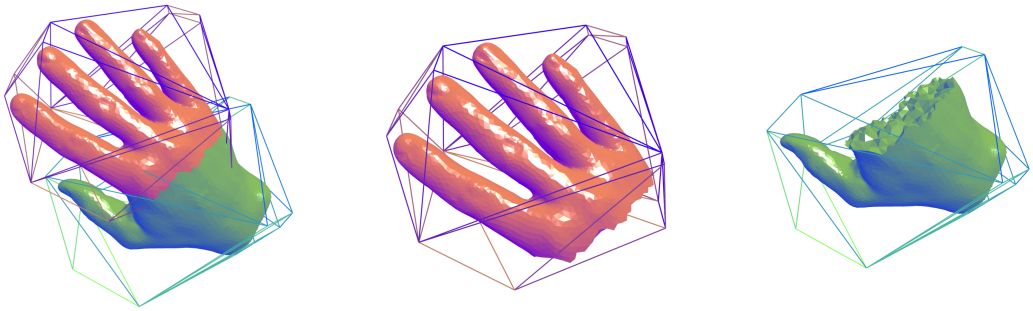
describes the grid-index at dimension k. The cost of evaluating the hash function h only depends on the dimension d. If we fix d, the computational time complexity of evaluating h is constant. We build for each cell of the grid a list L of possibly intersecting objects by exploiting h. This indexing process has $\mathrm{O}(n|L|_{\max})$ where $|L|_{\max}$ is the maximum number of objects pointed by a cell. In order to index a polytope E (*e.g.,* an element of a mesh) we use its bounding-box $\mathscr{B}$ for identifying which cells is intersecting. We compute $\mathscr{I}^m = \mathscr{I}(\mathbf{B}^m)$ and $\mathscr{I}^M = \mathscr{I}(\mathbf{B}^M)$ which are respectively starting and ending tensorial indices of a range of cells of the grid. We iterate over the cells within this range and we append E to the list of the corresponding table entry. Elements generally intersects more than one cell, hence when we compute the list of intersections for some element, we flag the elements that have been encountered in any of the visited cells, in order to avoid adding them twice in the intersection list. Once this list is complete we remove the flags from its elements.

The performance of the hash-grid is dependent on $|L|_{\max}$, which can grow (unnecessarily) in scenarios where there is high variability of element sizes and positions.

Hierarchical grids allow to treat data with different size more efficiently than simple uniform grids. A hierarchical grid is composed by a set of nested grids organized by levels. The main difference with space-partitioning trees (Section 2.4.4) is that there is no root. Hierarchical grids are extensively explained in [41].

2.4.4 Space-partitioning trees and bounding volume hierarchies

Binary space partitioning trees (BSP trees) recursively subdivide space into convex subsets. This subdivision is done by means of hyper-planes which can have any orientation. A special case of BSP trees, where the hyper-planes have the orientation of the canonical basis vectors, are *kd-trees*, *quadtrees* and *octrees*. The latter ones are used to partition space by recursively subdividing it with either four quadrants for the quadtree, or eight octants for the octree. From now on we refer to quadrants and octants as cells. This partitioning strategy allows for



Overview. *k*-DOP for one process. *k*-DOP for another process

Figure 2.8. The *k*-DOP is employed to efficiently discard trivial negatives for the intersection detection. The hand mesh is courtesy of Pierre Alliez, INRIA (Aim@Shape Project).

efficient spatial queries for finding intersecting/near geometric objects. The hierarchical structure is described by a set of nodes, each node is a cell, and it is either a branch, a leaf, or the root. A branch represents a subdivided cell, pointing to a set of sub-cells (children) which are either branches or leaves. A leaf represents the smallest cell, and usually stores the information related to the geometric data. The root, represents the top branch where the associated cell describes the whole volume of interest. A node storing no data, *i.e.,* no geometric data in the case of leaves, and no children in the case of branches, is referred to as an empty node.

Bounding volume hierarchies (BVHs). In BVHs the leaf nodes of the tree are the geometric objects, these geometric objects are usually wrapped in bounding-volumes. The leaves of the tree are grouped as small sets and enclosed within larger bounding volumes, which form the branches of the tree. BVHs can be constructed with different types of sub-volumes such as spheres, cubes, *k*-DOPs, *etc.*. The difference between BVHs and the other type of trees described in this section, is that the bounding-volume associated with the nodes at the same level do not have to form a partition.

2.4.5 Space-filling curves and linear octree/quadtree representations

Space-filling curves are a popular choice for data partitioning due to their proximity preserving properties. We refer to [6] and the literature cited therein for an extensive explanation of space-filling curves and their applications.

Linear octrees are widely used for collision detection [41] and for parallel

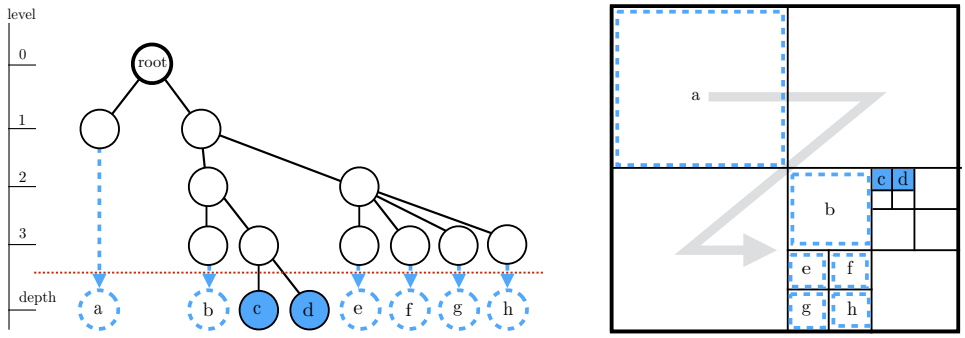


Figure 2.9. Left: tree representation of the quadtree. Below the dashed line we have the leaves which are considered in the ordered linear representation of the tree. Here the leaves are following a fractal z-curve. Right: a square domain decomposed using a quadtree. The shaded nodes are the leaves created at maximum depth, and the dashed nodes are leaves which are flattened in order to ensure uniqueness of the key associated with the leaves.

load balancing algorithms [129]. A linear octree consists of just the octree nodes which contain data. These nodes are one-dimensionally sorted such that their associated geometric data is ordered following a space-filling curve. The sorting is based on a particular choice of keys associated with the nodes. To generate the keys, or hashes, we choose the Morton encoding [41]. An example is depicted in Figure 2.9, where the nodes in the linear representation are ordered based on the Morton encoding. To generate the Morton keys, once the depth d is fixed, we have to consider all the leaves as they were at level d. In Figure 2.9 the dashed arrows represent the flattening and the dashed circles are the leaves which are now considered at the tree depth d. The Morton ordering is a particular choice of space-filling curve, however any other representation might be chosen for our purposes. Hence, for a node n, let $h_d(n)$ denote the Morton code (or key). The number of unique keys is usually equal to the maximum number of leaves of an octree of depth d.

2.4.6 Advancing front algorithms

As mentioned in Section 2.3, for handling the information transfer between two meshes, we could reach best-case linear computational time complexity by means of the advancing-front algorithm proposed in [49]. From a mesh $\mathcal{T}$ defined as a set of elements E and a set of nodes $\mathcal{N}$, we construct its element adjacency graph in linear time by finding elements with common nodes. This graph is used to find the intersections of two meshes $\mathcal{T}_m$ and $\mathcal{T}_s$ in linear time. We first

find a pair of intersecting elements $\{E_m, E_s\}$. We compute the intersection and determine if E_m is also a viable candidate for the neighbors of E_s. Then, we use the adjacency graph of E_m to test the neighboring elements for intersections with E_s. We repeat this operation until there are no elements intersecting E_s and we mark E_s as resolved, and we move to the neighbors of E_s and repeat.

In spite of the aforementioned advantages of this approach, we choose tree-search algorithms. Although the advancing-front algorithm in the best case has lower computational time complexity bound ($\mathrm{O}(n)$ instead of $\mathrm{O}(n\log(n))$, where n is the size of the input), it does have high hidden additional requirements in terms of computational cost. For instance, it requires information on what meshes or partitions need to be intersected with each other, and to determine each starting couple of intersecting elements. Particularly in parallel environment with arbitrarily distributed meshes this might not be trivial, or even not efficient. Additionally, with tree-search algorithms we can allow more use cases, as mentioned in the introduction of this thesis, without having to change our search strategy.

2.5 Parametrizations and finite element discretizations

The finite element method allows simulating physical phenomena while representing complex geometric objects by means of meshes. Such geometric objects are complex geometric descriptions which are generated by computer aided design (CAD) software, captured from real life objects or organisms (*e.g.,* 3D scans, MRI, *etc.*), and need to represented in a sufficiently accurate way. This is the case because the accuracy of the geometric representation influences the approximation error of the discrete solution of a partial differential equation.

The influence of accuracy of the geometric representation on the approximation error has been studied for curved boundary of iso-parametric discretizations [26; 114; 115] and for contact problems [75]. More recent literature focuses on numerical studies for different approximation spaces [15; 14], and elliptic and Maxwell problems [140].

During a simulation the approximation space might not be descriptive enough to represent the solution. This problem is usually solved by means of adaptive refinement strategies, such as h-refinement [16; 18] and p-refinement [95]. When using such strategies, a higher resolution at the boundary should be accompanied by a better approximation of the original surface [37]. However, due to the one-way connection between geometric information and simulation environment, the adaptive refinement is rarely accompanied by an increase in the

accuracy of the shape. In other words, the mesh is generated within a modeling software and used in simulation environments without considering the original surface, preventing a better surface approximation.

Iso-geometric analysis (IGA) [67] allows to overcome this limitation by working directly with the geometric description provided by CAD software, such as non-uniform rational B-splines (NURBS). However, IGA is subject to several challenges such as the treatment of non-watertight surfaces, local refinement and topological flexibility [87]. Moreover, IGA approximations, similarly to many mesh-free methods, leads to complications in the imposition of essential boundary conditions, which can be either imposed in a weak sense [11], or least-squares satisfied in the strong sense [67].

Additionally, when dealing with three-dimensional shapes, CAD models usually describe only the boundary. Creating a NURBS volume parameterization is a complex task, for which many different approaches exits. For instance, some of them require particular shapes [1], need special geometric information [93], or do not reproduce the surface exactly [85].

An alternative to IGA is the NURBS-enhanced finite element method (NEFEM) [124] that allows exploiting CAD geometries to describe exactly the boundary of the geometry. However, this method requires creating a parameterization mesh, and a special handling of the boundary, which according to [124] is still an open problem.

Another challenge regards geometric multigrid methods which require a hierarchy of nested spaces for optimal convergence [57; 22]. Such requirement can be satisfied by generating the hierarchy by refining a coarse mesh representation of the shape. However, such refinement cannot improve the shape accuracy. An alternative approach [35] is to employ a parameterization such as transfinite interpolation [110; 111] and to build nested hierarchies in the parameterization domain. However, transfinite interpolation requires a surface parametrization, a specific parametrization domain, and it is not guaranteed to be bijective for general polytopes.

2.5.1 Composite mean value mappings

To the best of our knowledge only the *composite mean value mappings* [121] allow creating smooth-bijective mappings between polytopes, such as polygons or polygonal meshes. Convenient properties of such mappings are that they can be evaluated point-wise, are provided in closed form, and are C^∞ in the interior

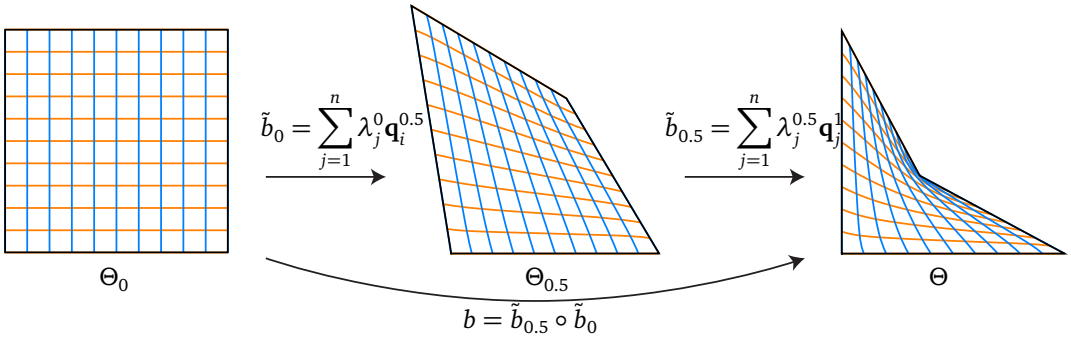


Figure 2.10. Overview a composite barycentric mapping for $\tau = [0, 0.5, 1]$.

of the domain. These mappings are based on the *mean value mapping*

$$\tilde{b}(\boldsymbol{x}) = \sum_{j=1}^{n} \lambda_j(\boldsymbol{x}) \mathbf{q}_j,$$

where $\mathbf{q}_j$ are the vertices of Θ and the functions $\lambda_j : \Theta_0 \to \mathbb{R}$, $j = 1, \ldots, n$ are a set of *mean value coordinates* [48] with respect to Θ_0. That is,

$$\lambda_j = \frac{w_j}{\sum_{k=1}^{n} w_k} \qquad \text{with} \qquad w_j = \frac{\tan(\alpha_{j-1}/2) + \tan(\alpha_j/2)}{r_j},$$

where α_j is the angle between the edges $[\boldsymbol{x}, \mathbf{q}_{j+1}^0]$ and $[\boldsymbol{x}, \mathbf{q}_j^0]$ and $r_j = \|\boldsymbol{x} - \mathbf{q}_j^0\|$, with $\mathbf{q}_j^0$ the vertices of Θ_0.

Unfortunately, the mapping $\tilde{b}$ is not guaranteed to be bijective for all pair of polytopes [70]. To overcome this limitation we follow [121] and "split" the mapping from source to target polytope into a finite number of steps, where each step perturbs the vertices only slightly. To this end, suppose that $\zeta_i : [0, 1] \to \mathbb{R}^2$, $i = 1, \ldots, n$ are a set of continuous *vertex paths* between each vertex $\mathbf{q}_i^0 = \zeta_i(0)$ and its corresponding vertex $\mathbf{q}_i = \zeta_i(0)$.

Let $\tau_s = (t_0, t_1, \ldots, t_s)$ with $t_k = k/s$ for $k = 0, \ldots, s$ be a *uniform partitioning* of $[0, 1]$ and $\tilde{b}_k$ be the barycentric mapping from Θ_{t_k} to $\Theta_{t_{k+1}}$, based on the barycentric coordinates $\lambda_i^{t_k} : \Theta_{t_k} \to \mathbb{R}$. Then we define the *composite barycentric mapping* from Θ_0 to Θ as

$$b = \tilde{b}_{s-1} \circ \tilde{b}_{s-2} \circ \cdots \circ \tilde{b}_0,$$

which is bijective for sufficiently large s according to [121]. Figure 2.10 shows an example of how a composite barycentric mapping is constructed for $s = 2$.

2.5.2 Efficient computation of the Jacobian matrix of the composite mean-value mapping

This section provides all derivations for efficiently computing the Jacobian $J_{\tilde{b}}$ of the 3D mean value mapping $\tilde{b}$ [73]. To compute $J_{\tilde{b}}$ we first need the formula for the partial derivative of the 3D mean value mapping of a point $\boldsymbol{x}$,

$$\tilde{b}(\boldsymbol{x}) = \sum_{i=1}^{n} \lambda_i(\boldsymbol{x}) \mathbf{q}_i^0 = \sum_{i=1}^{n} \frac{w_i(\boldsymbol{x})}{\sum_{k=1}^{n} w_k(\boldsymbol{x})} \mathbf{q}_i^0,$$

where w_i are the mean value weights [73] and $\mathbf{q}_i^0$ are the vertices of Θ^0. We first compute for each triangle $T = [\mathbf{q}_1^0, \mathbf{q}_1^0, \mathbf{q}_3^0]$ the quantities

$$d_j = \|\mathbf{q}_j^0 - \boldsymbol{x}\|, \qquad \mathbf{v}_j = \frac{\mathbf{q}_j^0 - \boldsymbol{x}}{d_j}$$

with gradients

$$\nabla d_j = \frac{-\mathbf{v}_j}{d_j}, \qquad J_{\mathbf{v}_j} = d_j \mathrm{Id} + \mathbf{v}_j (\nabla d_j)^T$$

for $j = 1, 2, 3$. If $\boldsymbol{x}$ lies on a vertex of the source polyhedron, then we know that the image of $\boldsymbol{x}$ is that vertex. Moreover, the function is only C^0 at the vertices, hence its gradient is not defined.

Then, we compute

$$r_j = \sqrt{4 - l_j^2}, \quad \theta_j = 2 \arcsin(l_j/2), \quad \nabla\theta_j = 2 \frac{(J_{\mathbf{v}_{j+1}} - J_{\mathbf{v}_{j-1}})(\mathbf{v}_{j+1} - \mathbf{v}_{j-1})}{l_j r_j}, \tag{2.16}$$

where $l_j = \|\mathbf{v}_{j+1} - \mathbf{v}_{j-1}\|$, $h = (\theta_1 + \theta_2 + \theta_3)/2$, and $\nabla h = (\nabla\theta_1 + \nabla\theta_2 + \nabla\theta_3)/2$. If $\boldsymbol{x}$ is inside T, then h is equal to π, and the image of $\boldsymbol{x}$ is given by the two-dimensional barycentric coordinates of that triangular face.

For efficiency reasons, we pre-compute

$$\begin{aligned}
s_{\theta_j} = \sin(\theta_j) &= \frac{l_j r_j}{2}, \quad c_{\theta_j} = \cos(\theta_j) = 1 - \frac{l_j^2}{2}, \\
s_{h_j} = \sin(h - \theta_j) &= \frac{l_1 l_2 l_3}{8} - \frac{l_j r_{j+1} r_{j-1}}{8} + \sum_{k=1, k\neq j}^{3} \frac{l_k r_{k+1} r_{k-1}}{8}, \\
s_h = \sin(h) &= -\frac{l_1 l_2 l_3}{8} + \sum_{k=1}^{3} \frac{l_k r_{k+1} r_{k-1}}{8}.
\end{aligned}$$

Note that the terms in the sums appear multiple times, hence we cache them. Instead of evaluating the cosines, we exploit the trigonometric identities to compute them from the sines, using exclusively square roots, which are much faster to compute. For instance, $\cos(*) = \gamma\sqrt{1-*^2}$, where the sign γ is computed by checking if the parameter h lies in the positive or negative region. We then compute

$$c_j = 2\frac{N_{c_j}}{D_{c_j}} - 1 = 2\frac{s_h s_{h_j}}{s_{\theta_{j-1}} s_{\theta_{j+1}}} - 1, \quad s_j = \operatorname{sign}(\det([\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]))\sqrt{1-c_j^2},$$

the corresponding gradients

$$\nabla c_j = 2\left(\frac{c_h \nabla h s_{h_j} + s_h c_{h_j}(\nabla h - \nabla\theta_j)}{D_{c_j}} - \frac{N_{c_j} c_{\theta_{j+1}} \nabla\theta_{j+1}}{s_{\theta_{j-1}} \sin^2_{\theta_{j+1}}} - \frac{N_{c_j} c_{\theta_{j-1}} \nabla\theta_{j-1}}{s_{\theta_{j+1}} \sin^2_{\theta_{j-1}}}\right),$$

and

$$\nabla s_j = -\frac{\nabla c_j \, c_j}{s_j}.$$

If $\boldsymbol{x}$ lies in the same plane as the triangle T and $\boldsymbol{x} \notin T$, then at least one of the three $s_j = 0$. In this case, T does not contribute to the computation of w_i and has to be skipped. Otherwise, we compute the mean value weight

$$w_j = \frac{N_{w_j}}{D_{w_j}} = \frac{\theta_j - c_{j+1}\theta_{j-1} - c_{j-1}\theta_{j+1}}{d_j s_{j+1} s_{j-1}}$$

and its gradient

$$\begin{aligned}\nabla w_j = &\frac{\nabla\theta_j - \nabla c_{j+1}\theta_{j-1} - c_{j+1}\nabla\theta_{j-1} - \nabla c_{j-1}\theta_{j+1} - c_{j-1}\nabla\theta_{j+1}}{D_{w_j}} \\ &- w_j\left(\frac{\nabla d_j}{d_j} + \frac{c_{\theta_{j+1}}\nabla\theta_{j+1}}{s_{\theta_{j+1}}} + \frac{\nabla s_{j-1}}{s_{j-1}}\right).\end{aligned}$$

Finally, we aggregate the local weights and gradients

$$w = \sum_{i=1}^{n} w_i, \quad \nabla w = \sum_{i=1}^{n} \nabla w_i,$$

and compute the Jacobian matrix of the barycentric mapping $\tilde{b}$,

$$J_{\tilde{b}} = \sum_{i=1}^{n} \mathbf{q}_i^0 \left(\frac{\nabla w_i + \tilde{b}_i \nabla w}{w}\right)^T.$$

The composite barycentric mapping and its gradient are computed in parallel for each evaluation point as

$$b = \tilde{b}_M \circ \ldots \circ \tilde{b}_2 \circ \tilde{b}_1, \quad \nabla b = \nabla \tilde{b}_M(\tilde{b}_{M-1} \circ \ldots \circ \tilde{b}_2 \circ \tilde{b}_1) \cdot \ldots \cdot \nabla \tilde{b}_2(\tilde{b}_1) \cdot \nabla \tilde{b}_1.$$

We modified the algorithm proposed in [73] in order to avoid computing trigonometric functions. In fact, our adaptation contains only the computation of the the arc-sine in (2.16), which is unavoidable because of the use of θ.

2.6 Software libraries and tools for scientific computing

As new technologies arise, scientific-computing software libraries need to be constantly updated or rewritten. For instance, the advent of GPGPU (general purpose graphics processing units) induced new programming paradigms and new languages such as CUDA [99] and OpenCL [74], which led to the creation of new software libraries such as cuBLAS [100] and ViennaCL [117]. Keeping up with such new technologies may cause small or significant changes in the code of applications such as non-linear solution strategies, finite element analysis, and data-analysis. However, the related high-level algorithms implemented in the application code should not have to change.

For this reason, one solution is to develop on top of a portable interface that fits many current and possibly future requirements (*e.g.,* PETSc [7] and Trilinos [58]). For instance, software libraries such as Deal.II [8], LibMesh [76], and MOOSE [50] rely on high level abstractions on top of existing linear algebra and non-linear solution strategies codes, and allow choosing the underlying implementation in a rather transparent way.

An alternative solution is exploiting scripting facilities for completely decoupling the application behavior from its actual implementation. This solution has the advantage of hiding the complexity of parallel software to which the average, casual or opportunistic [19], user is not supposed to be exposed. The idea is that the scripting code is translated to behavior which is implemented in another lower-level language. This enables users to write few lines of very powerful code without the overhead of learning how to use complex parallel scientific codes. A very specific form of scripting language is usually referred to as *domain specific language* (DSL). This specificity, while reaching the aforementioned objectives, has a twofold advantage. First, it enables a simple description of a specific problem since most implementation details can be hidden. Second, it allows exploiting complex functionalities and performance critical optimizations.

Notable examples related to finite element softwares, are FENICS's unified form language [88; 112], FREEFEM++ [71], and LISZT [31].

In DSLs lower-level abstractions are purposefully inaccessible because the actual algorithms are implemented in a different language, such as C++. This is a problem when a DSL misses a functionality, since adding it would require accessing the underlying back-end which may be either closed source or very complex. In contrast, embedded domain specific languages (eDSL) (*e.g.,* CULA [68], FEEL++ [104], OPENFOAM [135], SUNDANCE [90], VIENNAFEM [118]) uses the same language and compiler for both the "scripting" layer and the implementation of the back-end. For this reason, eDSLs have the opportunity to provide the right balance between abstraction and direct access to the back-end data-types and algorithms.

2.7 Chapter conclusion

In this chapter we first introduced the related work. We covered the variational transfer of discrete fields and provided detailed explanations on how to implement it for both volume coupling and contact problems in a serial environment. Then, we briefly touched the subject of volume parameterizations, their implications in finite element work-flows and our volume parameterization of choice: mean-value mappings. Finally, we provided an overview of the current trends in the development of scientific software. The topics we covered impact the workflow and structure of finite element softwares and open opportunities for simulating on more accurate geometric descriptions, more automation in coupling complex problems, and overall flexibility and usability.

Chapter 3

Parallel transfer of discrete fields for arbitrarily distributed unstructured finite element meshes

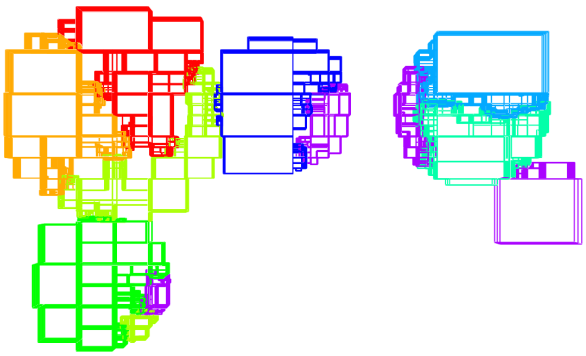


Figure 3.1. Parallel bounding volume hierarchy generated by our algorithm for detecting possible interections between arbitrarily distributed geometric objects. Color represents processes.

The algorithm presented in this chapter is an effort towards approaching the full automation of the geometric and functional coupling between different geometries and different approximation spaces in the context of coupled multiphysics simulations on complex geometries. Our parallel approach is flexible and can also be applied to different discretization techniques. And it is realized through a general software framework which does not require *ad-hoc* complex parallel code for each new scenario. In fact, our algorithm works under the as-

sumption that we might have multiple meshes arbitrarily decomposed, arbitrarily distributed, and their relation based on spatial information is not known.

However, it is imporant to note that in cases where the location information is trivially and globally accessible, for instance with Cartesian grids or structured meshes, our approach is not optimal. Our algorithm is designed as a general black-box solution, hence it might not be as efficient as a reduced variant specifically designed for Cartesian grids.

In Section 3.1, we provide an overview of the parallel algorithmic pipeline. In Section 3.2, we present a parallel spatial search algorithm and the necessary data-structures (Figure 3.1) for finding near or intersecting geometric objects, such as the mesh surface or volume elements. In Section 3.3, we present how the operator can be represented within the code, and how we can solve and handle all the relationships in one pass of the algorithm. In Section 3.4, we discuss what existing software tools we can use in order to implement the approach.

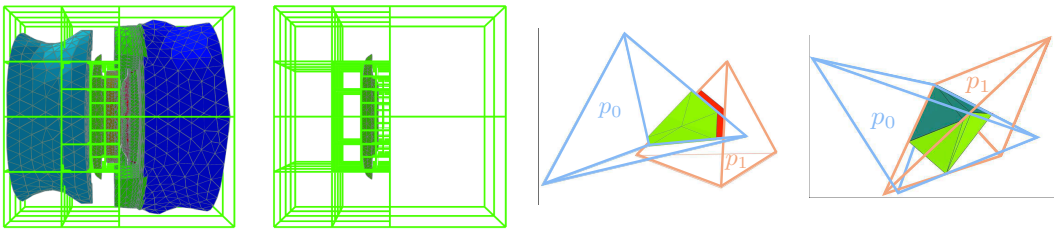
3.1 Parallel pipeline

The pipeline of our parallel approach to surface and volume projections consists of two main phases, a search phase and a compute phase. Each process, from its local knowledge about the data, gathers a minimal amount of information allowing it to determine its dependencies and executes the assembly of the transfer operator in a possibly balanced way. We will use the following terminology in order to refer to different roles that a process can have with respect to data and computation: *owner process*, which is a process owning a specific set of data before starting our algorithm and the related output at the end of our algorithm; *worker process*, which is a process performing computation on data which might or might not be its own. Our two-stages parallel pipeline can be summarized as follows:

1. *Parallel intersection/proximity detection*, explained in Section 3.2. A set of input meshes distributed arbitrarily to different owner processes is used as an input to our parallel tree-search algorithm. The tree-search algorithm then creates list of candidate-matching-element-pairs. This list of candidate pairs is used to identify near or intersecting objects (see Figure 3.2), and it is partitioned evenly among the processes for computation.

2. *Parallel operator assembly*, explained in Section 3.3. This phase is divided into a local and a global part. In the local part, we compute geometric projections/intersections and generate meshes on the intersections for the

quadrature; furthermore we assemble the entries of the coupling operators **D** and **B** and other application dependent quantities. In the global part, we create and communicate the complete representation of the transfer operator $\mathbf{T} = \mathbf{D}^{-1}\mathbf{B}$ from the worker process to the owner processes together with the other application specific quantities.



(a) A mesh of human vertebrae [51] and an octree data-structure.

(b) Detail of the spatial partitioning, with only the mesh of interest.

(c) Geometric surface projection of a tetrahedron facet.

(d) Intersection of two tetrahedral elements.

Figure 3.2. From a set of distributed meshes (a) we find the candidate-matching-element-pairs and their physical location (memory) for either surface projection (c), or volume projections (d). The elements are distributed in a grid of processes $\{p_i\}$, $i = 1, 2, \ldots, N$. The tree is constructed in parallel by exploring paths only with respect to the local geometric data, as shown by the example in (b).

3.2 Parallel intersection/proximity detection

The output of the intersection/proximity detection phase is a collection of candidate-matching-element-pairs $\langle E_m, E_s \rangle$, such that E_m can either be near or intersecting E_s. Both elements might be originally owned by any pair of processes, hence stored in two different memory spaces. The pair $\langle E_m, E_s \rangle$ however, once detected, is assigned to a worker process.

3.2.1 A parallel tree-search algorithm

The main goal of the search algorithm is to detect possible intersection candidates. The input consists of unrelated meshes, and an application related predicate. The predicate is used to determine if two meshes (also at the element level) and consequently two processes to which these meshes are assigned, need to be related or not. As previously introduced, the output consists of lists of

intersection-candidate pairs relating geometric objects (*e.g.,* tetrahedra , hexahedra) on pairs of processes. For sake of simplicity and clarity, let us assume that we have only one mesh or a subset of one mesh per process (for details about multiple meshes see Section 3.2.3). Efficient collision detection algorithms are usually divided into two phases, broad and narrow. In the broad phase, the tests are conservative and fast in order to reject trivial negatives. In the narrow phase, collision tests are exact and the actual intersection data is computed. We follow a similar structure to illustrate our strategy, hence we divide it in three main detection phases: broad, middle, and narrow.

Broad-phase detection

The main purpose of this phase is to eliminate, in the cheapest way possible, any trivial negative for our search, and the identification of which processes are related and which are not. In each process, we locally construct bounding volume data: an AABB and a k-DOP, as introduced in Section 2.4.1. We exchange among all processes the bounding volume data together with application predicate data. We are now able to discard trivial negatives and have pair-wise relations between processes. We call two processes related if they have a common non-empty partition of space accepted by the application predicate. With the union of all local AABBs, we can create a global AABB which will be the root of our tree search. Note that at this point we might already have created a sparse communication graph, hence allowing independent and specific point-to-point communication between related processes. A simple example is depicted in Figure 3.3. In Figure 3.3(a) we have one mesh per process, $\mathcal{T}_1$ for process p_1 and $\mathcal{T}_2$ for process p_2, and we want to determine the possible contact boundary between them. At the end of the broad-phase detection the overall knowledge of process p_1, as shown in Figure 3.3(b), consists of its local information, the global bounding box and the bounding box associated with process p_2 and $\mathcal{T}_2$. This information allows us to reduce the amount of data to be considered substantially. In case of more complex set-ups (*e.g.,* concave objects) this is not always the case. Hence, a refined search might be necessary, which brings us to the middle-phase detection algorithm.

Middle-phase detection

The main purpose of this phase is to detect, on a finer scale, which non-empty partitions of space exist and are shared between different processes. In order to do that, we build a lookup table, where each partition of interest is mapped to

all processes where this partition is non-empty. We realize this by performing multiple simultaneous breadth-first traversals of the tree (quadtree, octree, or n-dimensional generalization). More precisely, we perform a breadth-first traversal for each pair of related processes. A traversal must be simultaneous between a pair of related processes, however the other traversals in both processes can be considered concurrently, hence allowing for additional parallelism. The traversals can be performed by taking advantage of asynchronous communication (*e.g.,* when implementing the algorithm with the MPI standard), *i.e.,* by opportunistically advancing a traversal whenever new information is received by any other process.

We start with the partition of space described by the root of the tree, which is the same for every process (constructed in the phase described in Section 3.2.1). Local to each process p, we have to consider the following objects:

- For the nodes of the tree, a lightweight representation is needed in order to be exchanged after each iteration of the algorithm, and its essential form comes with the following data: the node-id; a first boolean flag, whose value is true when the node is empty and false otherwise; a second boolean flag, whose value is true if it is a leaf node and cannot be refined (hence changed into a branch), and false otherwise. The refinement of the search can be controlled by application specific predicates.

- A queue Q_{pq} for each related process q, where $Q_{pq} \equiv Q_{qp}$ (*i.e.,* equivalent). The queue $Q_{pq} = [n_0, n_1, \ldots, n_k]$ contains the next k nodes of the tree to be visited for the simultaneous traversal of the process pair $\{p, q\}$. In order to perform a breadth-first traversal, the queue is treated with a first in first out (FIFO) policy, hence we push to the back of the queue and pop from the front. With $Q_{pq}(i)$ we describe the ith element from the front of the queue.

- We define the lightweight representation of Q_{pq} as the queue $\tilde{Q}_{pq}$, whose elements are lightweight node representations. This representation is created by process p, and communicated to process q.

- A list L_{pq} of non-empty nodes, such that $L_{pq} \equiv L_{qp}$. These nodes are either leaves or branches, and identify where the various paths of the traversal stop.

Each process p performs the following operations at each level l for each related process q, $q \neq p$:

- If $l = 0$, we push the children of the root into queue Q_{pq}, and we continue to the next level.
- If $l > 0$, we create $\tilde{Q}_{pq}$ from Q_{pq} and we send it to process q, and we receive $\tilde{Q}_{qp}$ from q.
- We process each pair $n_i = Q_{pq}(i)$, $\tilde{n}_i = \tilde{Q}_{qp}(i)$, where $i = 1, \ldots, m$ and m is the number of elements in $\tilde{Q}_{qp}$ and $\tilde{Q}_{qp}$ (note that $\tilde{Q}_{qp}$ contains exclusively nodes at level l). If n_i or $\tilde{n}_i$ are empty, we move on to the next pair. If either n_i or $\tilde{n}_i$ are leaves and cannot be refined to branches, we add n_i to L_{pq}. Otherwise, if n_i is a leaf, then we refine n_i to a branch, and we push the children of n_i into Q_{pq}. All m nodes which we considered are popped from Q_{pq}. It can be observed that this part of the procedure is and needs to be symmetric with respect to p and q, ensuring consistent representations of the traversal between related processes.
- If Q_{pq} is empty, we end the simultaneous traversal with respect to the pair p and q. Additionally, if L_{pq} is also empty, we consider p and q to be unrelated.

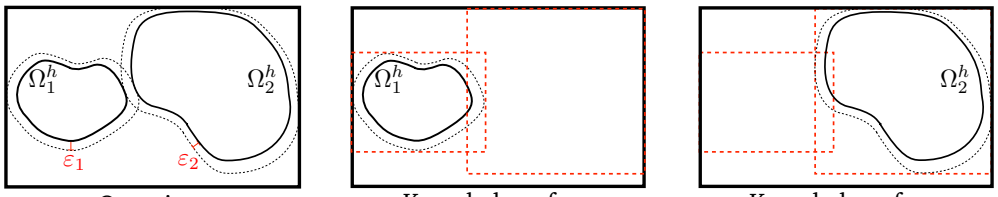
We can now construct the lookup table L_p from p to any related process q by gathering the information in all L_{pq}. For each node of the local tree we identify in which other processes the same (remote) node exists and how much data it contains. This information can be used to balance the narrow-phase search.

An example execution of the middle-phase detection algorithm is illustrated in Figure 3.4. Here, we can see how the quadtree structure is updated after each iteration, until we obtain a small partition of space containing all the data of interest.

Linearization and load-balancing

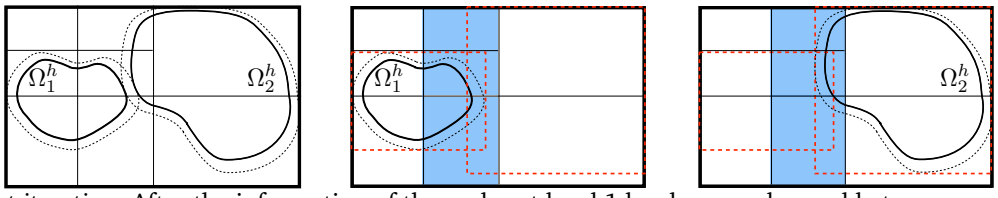
Generally, the middle-phase detection generates an output that gives rise to an unbalanced narrow-phase search. For this reason, before entering the narrow-phase detection phase, we redistribute the work. Hence, we first generate a linear representation of the tree with local Morton ordering, then we identify the pair-wise matching nodes by means of the lookup table resulting from Section 3.2.1, balance and redistribute the node and related data accordingly.

With the set S_p, we define the data storage (*e.g.,* the mesh) for each element $E \in S_p$ owned by process p. We now construct a re-purposed linear representation of the tree as a piecewise-ordered set of nodes Z. We recall the lookup table and its entries L_{pq} defined in Section 3.2.1. A node $n \in L_{pq}$ and a node $m \in L_{pr}$,

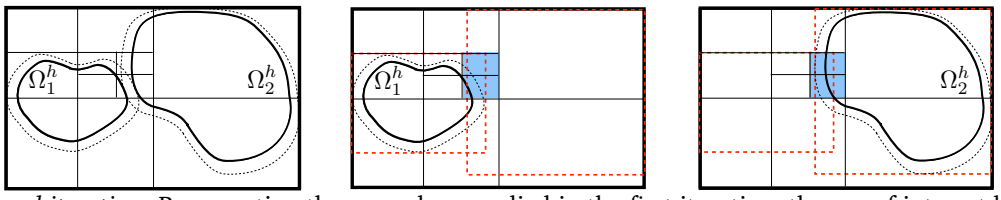


Overview. Knowledge of p_1. Knowledge of p_2.

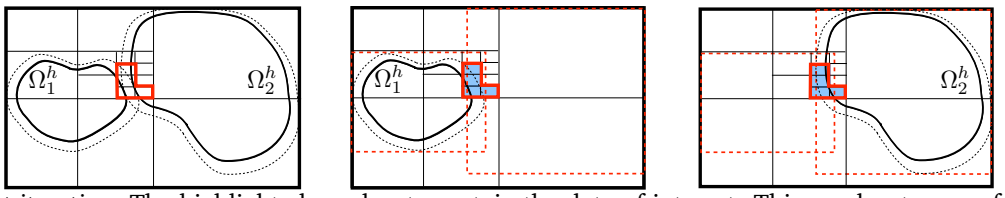
Figure 3.3. Parallel tree-search: result of the broad-phase detection algorithm. The objective is detecting near/intersecting elements between meshes $\mathcal{T}_1$ and $\mathcal{T}_2$. Here, we can see the respective polygonal domains Ω_1^h and Ω_2^h. owned respectively by process p_1 and p_2. The roots' bounding volumes and the user meta-data are exchanged. In case of surface projections, in order to detect near surfaces, the faces are considered to be blown-up in normal direction by a function $\varepsilon_i : \Omega_i^h \to \mathbb{R}$.



First iteration. After the information of the nodes at level 1 has been exchanged between p_1 and p_2, both processes know which partitions are of interest and which are not. The nodes in the shaded area will be refined, and the children exchanged.



Second iteration. By repeating the procedure applied in the first iteration, the are of interest has become smaller.



Last iteration. The highlighted quadrants contain the data of interest. This quadrants are referenced in the lookup tables $L_{1,2}$ and $L_{2,1}$ introduced in Section 3.2.1.

Figure 3.4. Parallel tree-search: the middle-phase detection algorithm. Local knowledge of a pair of processes. Left column: overview. Middle column: knowledge of p_1. Right column: knowledge of p_2. For each iteration, the area of interest, hence the focus of the search, is marked by the shaded area.

with $h_d(n) = h_d(m)$, hence with same key, are considered to be the same node if and only if $q = r$. Let B_n be the AABB of node n and B_E of element E, we say that n contains E if and only if $B_E \cap B_n \neq \emptyset$. Let $N_p(n)$, be the number of elements contained by node n for process p, hence $E \in S_p$ is counted if contained by n. Let $C(n) := C_{pq}(n)$ be a cost function for node n and the pair of processes p and q, for instance $C_{pq}(n) = N_p(n)N_q(n)$ or $C_{pq}(n) = N_p(n) + N_q(n)$. Since node n is both in L_{pq} and in L_{qp}, for each process p we create a new set, our unbalanced local work set

$$U_p = \bigcup_{p \leq q} L_{pq}.$$

Locally to each process p, for each node $n \in U_p$ we generate its key $h_d(n)$ and sort U_p according to the node keys, generating the ordered set $\hat{U}_p \leftarrow \text{sort}_{h_d}(U_p)$. For nodes with equal key, the rank of their associated remote process is used as secondary key, hence for $n, m \in \hat{U}_p$, with $n \in L_{pq}$ and $m \in L_{pr}$, such that $h_d(n) = h_d(m)$, n comes before m if $q < r$. Note that d is the depth of the global tree hence $d = \max_p(d_p)$ where d_p is the local depth for process p.

In a slight abuse of notation let us define the distributed piecewise-ordered set as a concatenation of the local ones, hence $\hat{U} = \hat{U}_1 \circ \hat{U}_2 \circ \ldots \circ \hat{U}_P$ where P is the number of processes, and $\circ$ is the concatenation operator. With $\hat{U}(i)$, where $i \in \{1, 2, \ldots, |\hat{U}|\}$, we define the i-th element of $\hat{U}$, hence the i-th node. With $X(j) = X(j-1) + C(\hat{U}(j-1))$, where $j \in \{1, 2, \ldots, |\hat{U}| + 1\}$ and $X(1) = 0$, we define the cumulative cost at node $U(j)$. With an exclusive scan operation we can compute X. We now know the total cost $C_T = X(|\hat{U}| + 1)$, average cost $C_A = \lfloor C_T / P \rfloor$ and remainder $C_R = \text{mod}(C_T, P)$.

Our goal is to distribute the nodes and their content such that each process p will have a balanced local work set Z_p such that the associated cost $C_{Z_p} = \sum_{n \in Z_p} C(n)$ is as near as possible to C_A.

For each process p in parallel:

- For each process q:
  - Compute lower bound l_q and upper bound u_q. If $q \leq C_R$ then

    $$l_q \leftarrow (q-1)(C_A + 1) \quad \text{and} \quad u_q \leftarrow q(C_A + 1),$$

    otherwise

    $$l_q \leftarrow (q-1)C_A + C_R \quad \text{and} \quad l_u \leftarrow qC_A + C_R.$$

  - If $l_q \leq X_p(i) \leq u_q$, $i = 1, 2, \ldots, |X_p|$ then append $\hat{U}_p(i)$ to Z_q^p. With Z_q^p we denote a local-to-p partial representation of the balanced work set Z_q, which will have to be sent to q in a second moment.

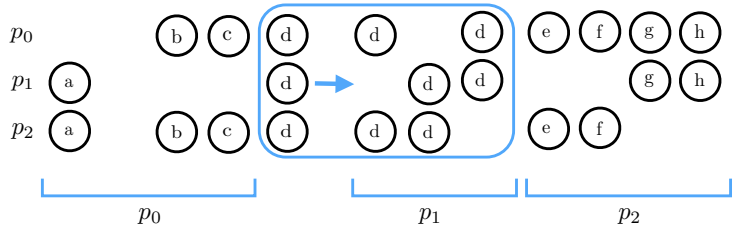


Figure 3.5. A circle represent a node of the linear tree. For each process p_i with $i \in \{1,2,3\}$, we see a row of nodes. Each row represent a local view of the linearized tree depicted in Figure 2.9. Each column represents matching node pairs. The work is partitioned along the space filling curve among the different processes p_i as shown by the grouping on the bottom.

Note that the data associated with each node $n \in Z_q^p$ is either on one or two different processes. With $D_{pr}(q)$, we describe the set of dependencies which are non local to p, owned by r and destined to the worker process q. This implies that in order to be able to completely construct Z_q, p has to send $D_{pr}(q)$ to r which then can construct Z_q^r. Now both owner processes p and r can send the correct data to the worker q to construct Z_q. Hence, we have now constructed $Z = Z_1 \circ Z_2 \circ \ldots \circ Z_p$. With $n \in Z_p$ we are able to directly access the data contained by n from process p.

Narrow-phase detection

The main purpose of this phase is to obtain the list of element pairs that are matching (intersecting), and related intersection data which is then to be used for quadrature as seen in Section 2.3. For each node of our linear tree we have two sets of elements which have to be matched with each other. We are searching matching-pairs at each node of the tree independently. A pair of matching elements might be detected in more than one node, and they might even be detected on different processes. Hence, in order to avoid redundant pairs we apply a simple selection rule. Let B be d-dimensional axis-aligned bounding-box (AABB) with minimum coordinates $\min_k(B)$, and maximum coordinates $\max_k(B)$, $k = 1,2,\ldots,d$. For a node n with AABB B_n and a pair of elements $t = \langle E_1, E_2 \rangle$ with respective AABBs B_1, B_2, the pair t is discarded whenever there exists a k such that $\min_k(B_1) \leq \min_k(B_n) \wedge \min_k(B_2) \leq \min_k(B_n)$. In order to avoid missing pairs at the boundary of the tree it is sufficient to enlarge the AABB at the root by a small value (*e.g.,* 10^{-6}). Note that, in order to avoind missing pairs of intersecting elements the AABB intersection test with the tree bounding boxes has to be

exactly as in (2.15) (*i.e.,* with the inequality operator). If the pair t is not discarded we perform our computation directly, or we first add the pair to a list of candidates associated with node n, then re-balance, and finally compute. For expensive computations such as the assembly of a transfer operator the second option is more effective. In this case, we can exploit the ordering of the nodes and re-balance the work by just reassigning the nodes such that the number of pairs is evenly distributed among the processes. If necessary the content of the node can be split to achieve a more fine-grained work partitioning.

3.2.2 Extended data-structures for pruning

Quadtrees and octrees might need several levels to reject non-intersecting data. In order to anticipate this rejection, we can use bounding volumes to provide a tighter bound for the content of each node of the tree. This bound will be added to the lightweight node representation introduced in Section 3.2.1, hence exchanged and tested against the remote counter-part, in order to prune the search. This might allow us to reduce the amount of edges in the communication graph in the first iterations of the middle-phase detection algorithm, mostly when handling complex shapes or chaotic distributions. Additionally, the elements in one node that we need to intersect can be tested against the bounding volume associated with the lightweight representation of the related remote node, in order to remove negatives before communicating. In other words, together with the octree, we are constructing a second bounding volume hierarchy (BVH) which is tightly describing the actual geometric data. One choice can be an AABB based BVH or a k-DOP based BVH. Though, the first choice is more efficient in the middle-phase detection it generates more false positives which might dramatically reduce performance in the computation phase.

3.2.3 Multiple meshes and multi-domain meshes per process

It is often the case to have multiple meshes per process (*e.g.,* geometric multigrid, contact problems, multi-physics problems, *etc.*). The application demands that we have multiple pairwise relationships between processes (*e.g.,* adjacent levels in multigrid hierarchies). In order to handle these set-ups, in a general way, we propose the following strategy: Each element is tagged with an ID representing a domain. When the element is inserted into the tree, the nodes encountered in the insertion paths are tagged with that same ID. A node might be tagged with multiple IDs. The lightweight node representation introduced in Section 3.2 also includes this information as a list of domain-IDs. Hence, at each node comparison

in the middle-phase detection algorithm we also check by using the application predicate if the local IDs and the ones received by the other process are related.

3.3 Application based assembly

We tackle any scenario in a monolithic fashion. That is, instead of assembling a separate matrix for each master-slave pair, where a projection needs to be computed, we assemble one single matrix describing all the different projections. Hence, there is one unique operator $\mathbf{T}$, which is assembled. The application predicates mentioned in Section 3.2 are provided by the user and are required to be able to discriminate between all domain (hence mesh) pairings. These predicates can be very simple, consisting only of the comparison of two integer numbers such as a domain identifier, or can be more elaborate depending on the user engagement and the application requirements.

We now describe the set-up and the projection matrix $\mathbf{T}$, assembled in this way, in detail. Given n discrete domains Ω_i^h with associated meshes $\mathcal{T}_i$ and $\overline{\mathcal{T}_i}$, $1 \leq i \leq n$, we assemble one matrix $\mathbf{T}$ containing all the different projection matrices $\mathbf{T}_{m,s}$ for every pair of intersecting meshes $\langle \mathcal{T}_m, \overline{\mathcal{T}_s} \rangle$ and related projection operator $P_{s,m} : V_h^m(\mathcal{T}_m) \to W_h^s\left(\overline{\mathcal{T}_s}\right)$, $1 \leq m, s \leq n$, Here, in order to also include the case of remeshing of multiple domains, two meshes $\mathcal{T}_i$ and $\overline{\mathcal{T}_i}$ are associated with every domain Ω_i^h. If no remeshing is done, the two meshes are the same, *i.e.,* $\overline{\mathcal{T}_i} = \mathcal{T}_i$.

The global projection matrix $\mathbf{T} = \mathbf{D}^{-1}\mathbf{B}$ is then the block matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{1,1} & \mathbf{T}_{1,2} & \cdots & \mathbf{T}_{1,n} \\ \mathbf{T}_{2,1} & \mathbf{T}_{2,2} & \cdots & \mathbf{T}_{2,n} \\ \vdots & & \ddots & \vdots \\ \mathbf{T}_{n,1} & \mathbf{T}_{n,2} & \cdots & \mathbf{T}_{n,n} \end{bmatrix},$$

where every block $\mathbf{T}_{j,i}$ is the matrix representation of a projection $P_{j,i} : V_h^i(\mathcal{T}_i) \to W_h^j\left(\overline{\mathcal{T}_j}\right)$. It maps a vector $\mathbf{v} = \left[\mathbf{v}_i\right]_{i=1}^n$, where $\mathbf{v}_i$ is the coefficient vector of a function in $V_h^i(\mathcal{T}_i)$, to a vector $\mathbf{w} = \left[\mathbf{w}_i\right]_{i=1}^n$, where $\mathbf{w}_i$ is the coefficient vector of a function in $W_h^i\left(\overline{\mathcal{T}_i}\right)$. Depending on the geometric set-up and application considered, the various blocks $\mathbf{T}_{j,i}$ of the operator $\mathbf{T}$ and of the vectors $\mathbf{v}$ and $\mathbf{w}$ might be zero or even undefined. Thus, even though the transfer problem $\mathbf{w} = \mathbf{T}\mathbf{v}$ seems to be a dense system in the monolithic form, in practice it is typically sparse. For numerical implementation, $\mathbf{T}$ might also be employed through its two separate components $\mathbf{D}^{-1}$ and $\mathbf{B}$ (*e.g.,* in case the inverse of $\mathbf{D}$ is too expensive to compute directly).

As previously introduced in Section 2.1, there is a wide range of applications suitable to our approach. All the representations of the operators related to these applications can be described by our monolithic representation. For instance, the ensemble of interpolation operators $\mathbf{T}$ and consequently restriction operators $\mathbf{T}^T$ (the transpose of $\mathbf{T}$). for a three-level geometric multigrid hierarchy is represented as follows

$$\mathbf{T} = \begin{bmatrix} 0 & \mathbf{T}_{1,2} & 0 \\ 0 & 0 & \mathbf{T}_{2,3} \\ 0 & 0 & 0 \end{bmatrix},$$

where $\mathbf{T}_{i,j}$ is the interpolation matrix from level i to level j with corresponding restriction matrix $\mathbf{T}_{i,j}^T$. For the transfer of state variables when remeshing n different geometric objects the interpolation operator $\mathbf{T}$ would be represented as follows

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{1,1} & 0 & \dots & 0 \\ 0 & \mathbf{T}_{2,2} & 0 & 0 \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \mathbf{T}_{n,n} \end{bmatrix},$$

where $\mathbf{T}_{i,i}$ is the transfer from the old version $\mathscr{T}_i$ to the new version $\overline{\mathscr{T}_i}$ of the mesh for Ω_i^h.

The examples are of course to be considered in the context of parallel computing and distributed memory. A simple scenario would consist of the communication graph matching by the non-zero block structure of $\mathbf{T}$, where each block $\mathbf{T}_{i,j}$ allows us to transfer quantities from process j to process i. Once the local contributions to $\mathbf{T}$ are computed, the entries are redistributed according to the original ownership of the entries, and added to the correct blocks. At this point, any parallel linear algebra library (*e.g.,* PETSc or TRILINOS algebra modules) can be adopted for applying the operator.

3.3.1 Element-wise block operator representation

A convenient option is to assemble the coupling operator $\mathbf{B}$ as a block matrix, where each block is associated to one element and it is disconnected from any other block. In other words a node has a unique degree of freedom for each incident element. Let us denote this variant of the coupling operator as $\tilde{\mathbf{B}}$. Once we have $\tilde{\mathbf{B}}$, we can obtain $\mathbf{B}$ by introducing $\mathbf{P}$, which allows us to compute

$$\mathbf{B} = \mathbf{P}^T \tilde{\mathbf{B}} \mathbf{P}.$$

The matrix **P** may come from discontinous Galerkin methods [3] or it might simply represent an aggregation from the disconnected nodal degrees to coupled nodal degrees of freedom. The same reasoning may be applied to **D** when necessary. The discontinuous operator representation is necessary when we need to discard parts of the operator as for instance in contact simulations as explained in Section 3.3.2. This representation might be convenient in the context of transient simulations, where the computational domain has both moving parts and static parts. In order to save computational time we can exclusively re-compute the operator for the moving parts.

3.3.2 Handling of assembled quantities in contact problem

The assembly of the transfer operator in the context of contact problems requires special handling of the assemble quantities at the boundary of the contact surface. Here, the intersections do not necessarily always match the surface of elements of the slave side (*i.e.,* there are case where the master surface does not cover the slave surface). This issue can be detected only after computing intersections of each slave elements with all intersecting elements on the master side, which in parallel executions might be performed by different processes.

In order to discard invalid contributions, we first have to communicate the quantities related to each surface slave element to the owner processes. We can sum up the entries of the coupling matrices entries associated with the element to compute the area of the intersection. If the intersection area is less than the area of the slave element we discard all its associated quantities. Once this selection has been performed we can build the actual transfer matrix, the gap vector, and the normal-tangential orthogonal transformation matrix introduced in Section 2.3.1. The self-contact algorithm for parallel scenarios is not resolved in this thesis.

3.4 Implementation

We implemented the whole algorithmic pipeline as part of the MOONOLITH library available at `http://moonolith.inf.usi.ch`. Though some parts of the algorithm might be suitable for hybrid parallelism, we restricted ourselves to a plain MPI (Message Passing Interface) implementation. The user can specify a domain level predicate for pruning the search in order to be more efficient, and relate domains and subdomains with each other. The user can also specify element-level predicates to avoid unwanted element matches. The user needs

to mark which element is a slave (or non-mortar) for handling the data dependencies for the assembly. If all elements are marked as slaves, then the assembly function is called for every intersecting element-pair (hence more expensive computation), and the user can decide what to do at the last moment.

For applying the transfer operator, hence computing the actual information transfer, we use the PETSc library.

In the implementation of this algorithmic framework within the MOONOLITH library the following information transfer scenarios are supported: the transfer of functions from a set of volume meshes to another, either in $2D$ or in $3D$; the transfer of functions from a set of surfaces to another in $3D$, optionally including the generation of contact surface data for contact problems between multiple elastic bodies, such as weighted gap functions, normals, and other user extensions; detection and balancing in n-dimensions.

Given that the MOONOLITH library has been implemented following object oriented programming principles, additional extensions can be easily added.

We integrated our parallel algorithmic framework with the MFEM library [77] for allowing volume transfer between all their available discretizations, and it is officially available as an optional module here `http://www.github.com/mfem`. We have implemented a full LIBMESH [76] integration both for surface and volume transfer available here `http://www.bitbucket.org/zulianp/utopia`.

3.5 Chapter conclusion

We presented a parallel approach for the assembly of transfer operators in the context of finite element simulations. These operators are of interest for a broad range of applications such as multi-physics simulations, non-conforming domain decomposition, contact problems, multi-scale simulations, and re-meshing. We focused our study on arbitrarily distributed unstructured meshes and the transfer of discrete fields with respect to volume and surfaces.

We introduced the approach in relation to the assembly of projection operators like the L^2-projection and its local approximations, nevertheless it can be employed for classical interpolation methods as well. We presented an algorithmic framework and at an abstract level also the implementation of the MOONOLITH library. This framework can be employed for handling 2D and 3D geometries with respect to both surface and volume geometries. The approach can also be employed for the case of surface projections in contact problems for computation of bounded distances.

Our approach is not optimal for the scenario when either the master or the

slave mesh is a structured-grid. In this scenario using spatial hashing algorithms is the most efficient variant even in parallel. Spatial hashing is rather simple to parallelize since the grid information can be fully replicated and stored by each process with a relative small memory occupancy. This replication allows to neglect complex communication routines.

Chapter 4

Parametric finite elements with bijective mappings

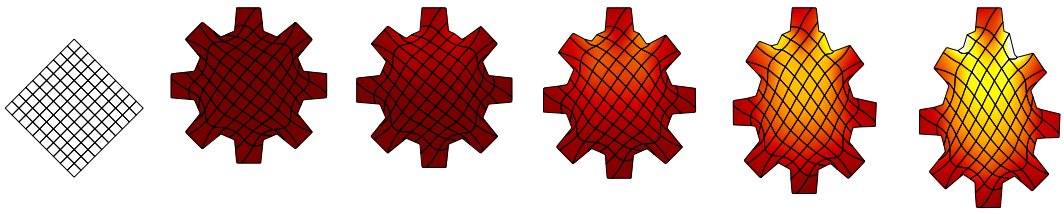


Figure 4.1. Transient non-linear elasticity simulation for a warped quad-mesh with compressible-neo-Hookean material. The elastic gear is subject to vertical body forces (gravity) and has a fixed tooth on the top boundary. The colour represents the von Mises stress for the solution at the different time-steps t.

In this chapter, we present a novel discretization which enables exploiting exact geometric descriptions (*e.g.,* splines or surface meshes) together with strategies employed in standard finite element simulations (Section 4.1). This discretization has the advantage of decoupling the geometry and the approximation space allowing for sub/iso/super-parametric elements. Although our presentation is based on the Poisson problem, our discretization can be naturally employed to solve more complex problems, such as transient non-linear elasticity shown in Figure 4.1.

Similarly to IGA, our approach focuses on a parametric representation of the input geometry. Despite this similarity we can discern the two techniques for the way they relate to standard finite elements, the existing codes, and the type of

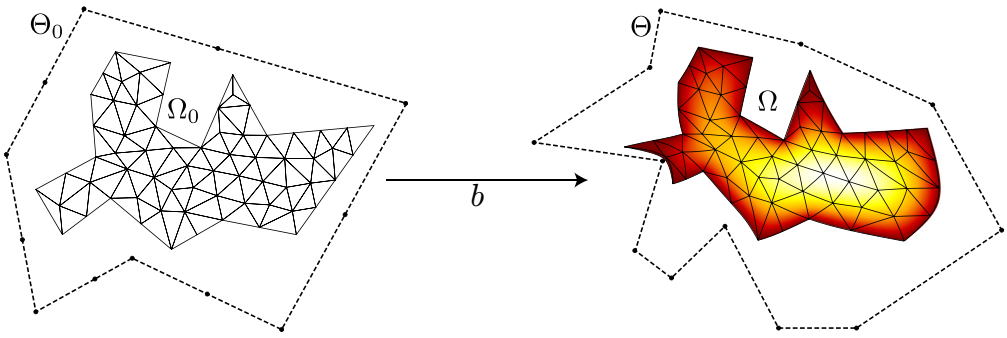


Figure 4.2. Overview of parametric finite elements with bijective mappings, with colour-coded solution of the Poisson problem (4.1) on a 2D warped domain Ω, with zero boundary conditions and constant right-hand side.

input geometries they handle. For our method the choice of the basis functions is not determined by the choice of the geometric mapping, whereas for IGA it does. The extension of standard finite element codes with the techniques described in this chapter is rather straightforward as it is illustrated in Section 4.1. The problem of dealing with exact geometries has been deeply studied for CAD geometries by the IGA community. Unfortunately, a similar study for surface meshes is missing. For this reason, we focus on the *exact representation* provided by surface meshes, and present the construction of a bijective volume parameterization from arbitrarily shaped domains to arbitrarily shaped meshes (Section 4.2).

4.1 Formulation

Let us consider the standard Poisson problem

$$-\Delta u = f, \qquad u|_{\partial\Omega} = g, \tag{4.1}$$

where Ω is the *computational domain*, $\partial\Omega$ is the boundary of Ω and g describes the boundary values. In contrast with the classical construction

$$\Omega = b(\Omega_0) \subseteq \Theta$$

is given by the image of a sufficiently smooth *bijective mapping*

$$b\colon \Theta_0 \to \Theta,$$

where $\Omega_0 \subseteq \Theta_0$ is a *source domain*, $\Theta_0 \subset \mathbb{R}^d$ is a *parameterization domain*, and $\Theta \subset \mathbb{R}^d$ is a *parameterization image*. Figures 4.2 and 4.3 show an overview of our construction and the solution of the Poisson problem (4.1).

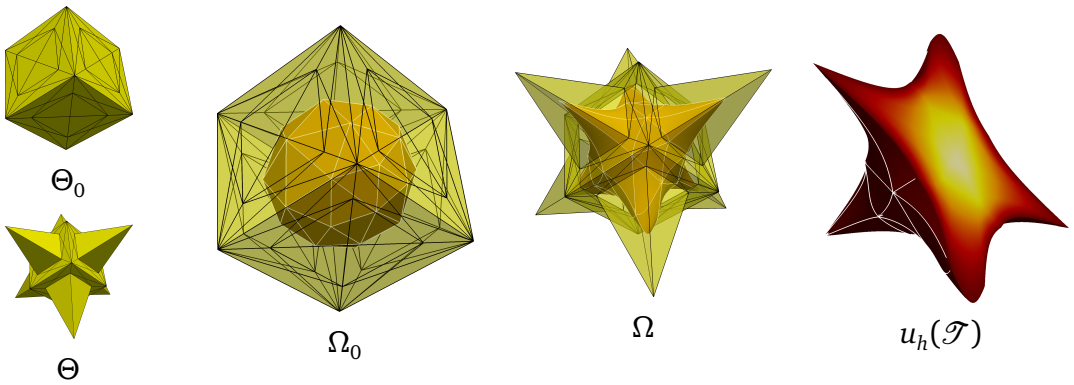


Figure 4.3. Overview of the parametric finite elements with bijective mappings, with colour-coded solution of the Poisson problem (4.1) on a 3D warped domain Ω, with zero boundary conditions and constant right-hand side.

Let $u \in V = H_0^1(\Omega)$, where H_0^1 is the Sobolev space of weakly differentiable functions vanishing on the boundary, and $f, g \in L^2(\Omega)$. Using integration by parts, we rewrite (4.1) in its weak form, which is: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v \qquad \forall v \in V.$$

Using b, we express the previous integral with respect to the *source domain* Ω_0. Considering that $u(\boldsymbol{x}) = u(b(\boldsymbol{x}_0))$ and $v(\boldsymbol{x}) = v(b(\boldsymbol{x}_0))$ where $\boldsymbol{x} \in \Omega$ and $\boldsymbol{x}_0 = b^{-1}(\boldsymbol{x}) \in \Omega_0$, and applying change of variables in the integrals, we rewrite the weak form: find $u \in V$ such that

$$\int_{\Omega_0} J_b^{-T} \nabla u \cdot J_b^{-T} \nabla v \ \det(J_b) = \int_{\Omega_0} f v \ \det(J_b) \qquad \forall v \in V, \tag{4.2}$$

where J_b is the Jacobian matrix of the mapping b.

In order to solve this problem, we represent the computational domain Ω by a warped mesh $\mathscr{T} = b(\mathscr{T}_0)$, where $\mathscr{T}_0 = \{E_0 \subseteq \Omega_0 | \bigcup \bar{E}_0 = \bar{\Omega}_0\}$ is a conforming mesh (*i.e.,* the intersection of pairs of different elements $\bar{E}_0$ is either empty, a common node, edge, or side) describing the source domain Ω_0 and the elements E_0 form a partition. Note that, as described in (4.2), the bijective mapping warps the entire volume, creating warped elements $E = b(E_0)$. Let the finite element space associated to $\mathscr{T}$ be

$$X_p^b(\mathscr{T}) = \{v \in C^0(\Omega) | \forall E \in \mathscr{T} \ \exists w \in \mathbb{P}_p : v(b(G(\hat{\boldsymbol{x}})) = w(\hat{\boldsymbol{x}}), \forall \hat{\boldsymbol{x}} \in \hat{E}\}, \tag{4.3}$$

abbreviated as X_p^b, where G the transformation from the reference element $\hat{E}$ to the corresponding element E_0 in the source domain, and $\mathbb{P}_p$ a space of polynomial

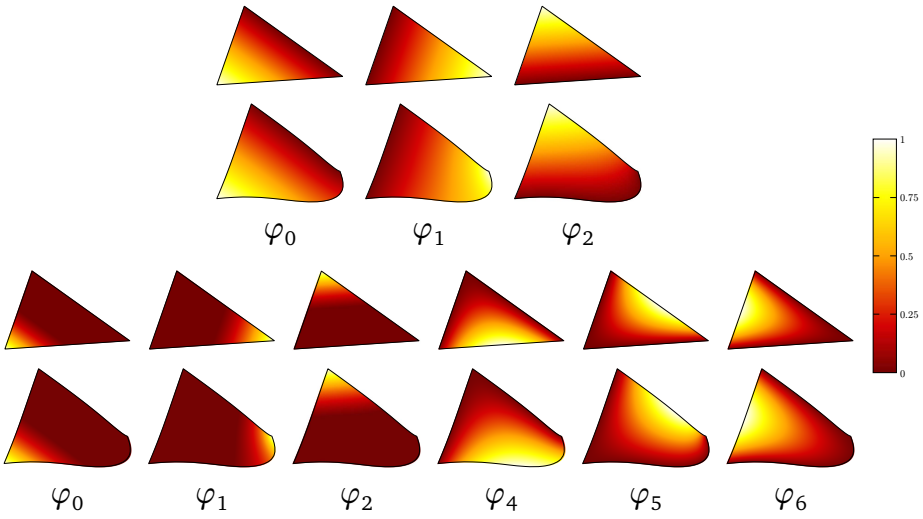


Figure 4.4. The standard linear and quadratic shape functions φ_i on the element of the *source mesh* and the corresponding warped element.

of order p defined in the reference element. Let the basis of X_p^b be $\{\varphi_1, \ldots, \varphi_m\}$, where m is the number of basis functions. Figure 4.4 depicts an example of such basis functions for a warped element. We approximate the function u by means of $u_h \in X_p^b$, where h stands for the discretization parameter. Expressing u_h in terms of its basis reads as $u_h = \sum_{i=1}^{m} u_i \varphi_i$, where u_i are real coefficients. By choosing the test space as X_p^b, we discretize (4.2) as

$$\sum_{i=0}^{m} u_i \int_\Omega J_b^{-T} \nabla \varphi_i \cdot J_b^{-T} \nabla \varphi_j \ \det(J_b) = \sum_{i=0}^{m} f_i \int_\Omega \varphi_i \varphi_j \ \det(J_b) \qquad \forall j = 1, \ldots, m,$$

which can be represented in the classical matrix form

$$\boldsymbol{L}\boldsymbol{u} = \boldsymbol{M}\boldsymbol{f}, \tag{4.4}$$

with $\boldsymbol{u} = [u_1, \ldots, u_m]^T$ and $\boldsymbol{f} = [f_1, \ldots, f_m]^T$

To assemble the Laplace stiffness matrix $\boldsymbol{L}$ and the mass matrix $\boldsymbol{M}$ we perform numerical quadrature. Because of the non-linearity of J_b we need to choose a proper quadrature scheme even when the basis functions of the approximation space X_p^b are low order polynomials.

We perform the quadrature in $\hat{E}$, using quadrature points $\hat{\boldsymbol{x}}_k \in \hat{E}, \boldsymbol{x}_k = G(\hat{\boldsymbol{x}}_k)$ with the respective *quadrature weights* $\alpha_k \in \mathbb{R}$, $k = 1, \ldots, K$. Figure 4.5 shows all the geometric transformations from the reference element $\hat{E}$ to the warped element E. We denote by $\hat{\varphi}_i$ the basis functions on the reference element and by

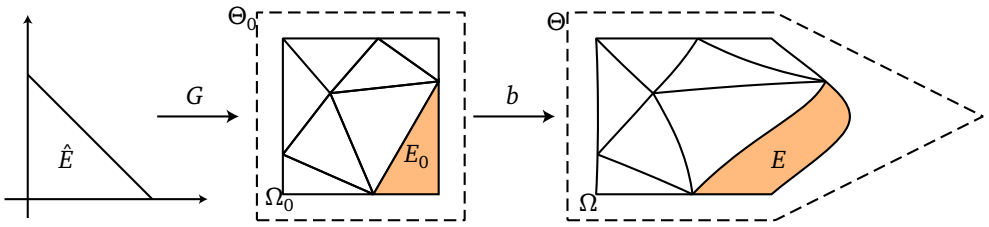


Figure 4.5. Overview of the geometric transformations from the reference element $\hat{E}$ to the source element $E_0 \in \mathscr{T}_0$ and to the warped element $E \in \mathscr{T}$.

J_G the Jacobian of G. This allows assembling the local matrices for the element E

$$\begin{aligned} L^E_{i,j} &= \sum_{k=1}^{K} \beta_k \, J^{-T}(\boldsymbol{x}_k) \, \nabla\hat{\varphi}_i(\hat{\boldsymbol{x}}_k) \cdot J^{-T}(\boldsymbol{x}_k) \nabla\hat{\varphi}_j(\hat{\boldsymbol{x}}_k), \\ M^E_{i,j} &= \sum_{k=1}^{K} \beta_k \, \hat{\varphi}_i(\hat{\boldsymbol{x}}_k) \, \hat{\varphi}_j(\hat{\boldsymbol{x}}_k), \end{aligned} \tag{4.5}$$

where $J(\boldsymbol{x}_k) = J_b(\boldsymbol{x}_k) J_G(\hat{\boldsymbol{x}}_k)$ and $\beta_k = \alpha_k \det(J(\boldsymbol{x}_k)) \left|\hat{E}\right|$, with $\left|\hat{E}\right|$ the volume of $\hat{E}$. These local contributions are then gathered to compute the matrices $\boldsymbol{L}$ and $\boldsymbol{M}$.

Note that the weak formulation and the assembly procedures are very similar to classical finite elements. In fact, the only difference is the usage of the geometric terms depending on the bijective mapping b, such as J_b which contributes to $J = J_b J_G$. As in standard FEM, the choice of the basis is independent from the geometric description, leading to super/sub/iso-parametric approximations. In our method the geometric description is given by the mapping b, which is usually non-linear, so that our discretization falls into the category of super-parametric elements.

If we assume that $b(\mathscr{T}_0)$ describes the exact geometry, then the geometric error is zero. However, the error in the solution is also connected to the choice of the approximation space and the shape of the elements. This error is influenced by the Jacobian J_b of the bijective mapping. We estimate it by means of the condition number

$$\kappa = \sup_{\boldsymbol{x}_0 \in \Omega_0, \boldsymbol{x} \in \Omega} \|J_b(\boldsymbol{x}_0)\| \, \|J_b^{-1}(\boldsymbol{x})\| \tag{4.6}$$

as in standard parametric finite elements estimates [17; 20].

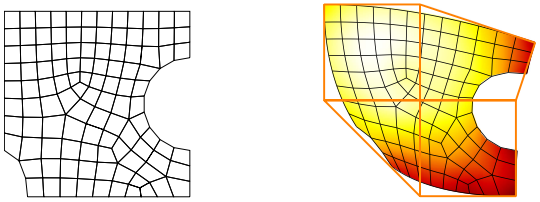


Figure 4.6. Example of parametric finite elements using a B-spline as the parameterization b. The colour describes the solution of the Poisson problem (4.1).

4.2 Shape and volume parameterization

The quality of a numerical solution of a partial differential equation is influenced by the accuracy of the geometric description and by the choice of the approximation space. In other words, a parameterization which describes the geometry exactly does not introduce any error related to the shape. The choice of this parameterization depends on the input geometry and includes every smooth bijective mapping, such as bijective spline mappings [42] (see Figure 4.6), composite mean value mappings [121], or harmonic mappings [120].

Since for CAD geometries the problem has been widely studied by the IGA community, we focus our study on volume parameterization between arbitrary surface meshes. The first challenge is the construction of a simpler surface Θ_0, a coarse *source domain* Ω_0, and a *paramaterization image* Θ, such that $\Omega = b(\Omega_0)$ (see Section 4.2.1). The other challenges are the construction of the volume parameterization b (see Section 2.5.1), and the efficient evaluation of the forms within a simulation work-flow (see Section 4.2.2).

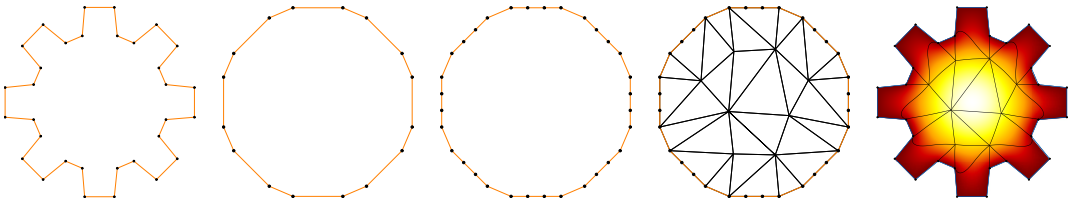


Figure 4.7. Given an input surface Θ we simplify it to obtain Θ_0, which coincides with Ω_0. We mesh Ω_0 obtaining $\mathscr{T}_0$ and solve the problem with respect to $\mathscr{T}$, which has the same boundary as Θ.

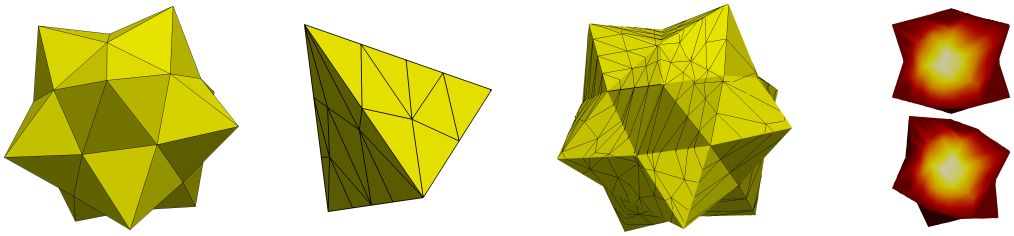


Figure 4.8. Three-dimensional example of the work-flow of our approach, from the input surface to the solution of the problem in the warped mesh $\mathscr{T}$.

4.2.1 Constructing the parameterization domain

In order to solve the model problem with the exact input geometry, the shape of Θ must coincide with Ω, which describes the exact shape. As carried out in detail in Section 4.1, our approach still requires a *parameterization domain* Θ_0 and a *source domain* Ω_0. Hence, we first need to construct Θ_0 with the same mesh connectivity as Θ while ensuring that Θ_0 describes a simpler shape. Note that in order to reproduce Ω by means of b the shapes of Ω_0 and Θ_0 must also coincide.

The approximation space for the finite element solution for the model problem can now be chosen independently from the shape, since Ω_0 and Θ_0 are arbitrary (*e.g.,* the octagon in Figure 4.7 or the tetrahedron in Figure 4.8). This allows meshing Ω_0 with arbitrary mesh size to obtain $\mathscr{T}_0$. Hence, by applying b to $\mathscr{T}_0$, we control the resolution of $\mathscr{T} = b(\mathscr{T}_0)$ independently from the shape of Θ without influencing the shape accuracy.

As illustrated in Figure 4.7, in the 2D case, Ω_0 is constructed by removing vertices from Θ. In order to obtain Θ_0 we reintroduce the removed vertices on the edges of Ω_0, without modifying the shape described by Ω_0. Finally, we mesh Ω_0 to obtain $\mathscr{T}_0$ and solve the problem in $\mathscr{T} = b(\mathscr{T}_0)$.

The 3D case requires to coarsen Θ in order to obtain Ω_0 while constructing a surface parameterization to build Θ_0 [40; 83; 80]. In our implementation we use the multi-resolution adaptive parameterization of surfaces (MAPS) algorithm [83], which produces a geometrically non-conforming parameterization (*i.e.,* Θ_0 is not nested inside Ω_0). To overcome this limitation, we extend the MAPS algorithm by snapping the vertices of Θ_0 to the edges of Ω_0, and by applying few element splits to Θ_0 and Θ when that is not feasible. We remark that the only operation performed on Θ is splitting, which does not change its shape.

Summing up, we start with a detailed mesh representing the exact geometry

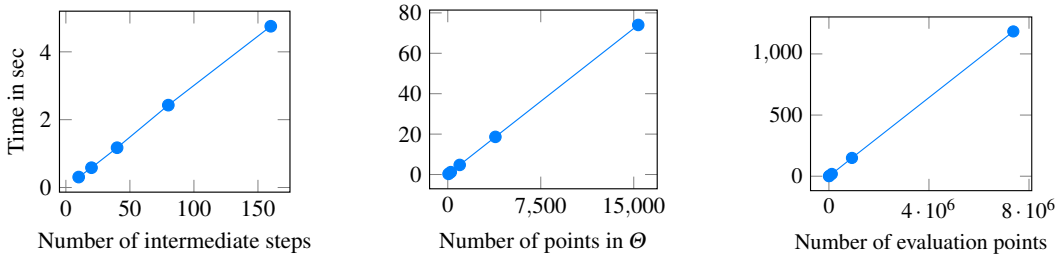


Figure 4.9. Running times for computing the composite mean value mapping and its Jacobian. The computational time depends on three parameters: the number of intermediate steps s, the vertices n of Θ, and the evaluation points. For each of the three experiments we vary only one of the parameters, whose base values are $s = 10$, $n = 62$, and 1800 evaluation points.

Θ. Then, from Θ we compute a coarse surface Ω_0 which we mesh to obtain $\mathcal{T}_0$. Finally, we use the parameterization obtained with MAPS to construct a surface Θ_0 with the same connectivity as Θ and the same shape as Ω_0. An example of a result of this procedure is shown in Figure 4.8.

4.2.2 Pre–computation of the composite mean value mapping

The composite mean value mapping described in Section 2.5.1 is computationally intensive. For this reason we need to avoid computing the mapping and its Jacobian multiple times. Similar to the classical assembly procedure of the problem matrices, we start by deciding the order of quadrature. The order of quadrature depends on the problem we want to solve, the choice of the approximation space, and, especially for our approach, the bijective mapping b.

Instead of directly assembling the matrices in (4.4), we divide the assembly procedure into two stages. The first stage consists of generating and storing all the quadrature data associated with the geometry necessary for the assembly, such as the global quadrature points $b(G(\hat{\boldsymbol{x}}))$ and the Jacobian matrices $J_b(\hat{\boldsymbol{x}})$.

The second stage consists of the standard assembly procedure of the element matrices (4.5), though using the precomputed quadrature quantities. This strategy allows assembling the matrices like for standard finite elements without the need of evaluating b and J_b for each new operator.

For the standard finite element assembly procedure storing the quadrature data is usually not necessary, making our two stage approach less memory-efficient. However, the caching allows both a parallel evaluation of b and the possibility of reusing the quadrature data for different operators (*e.g.*, Laplacian and mass

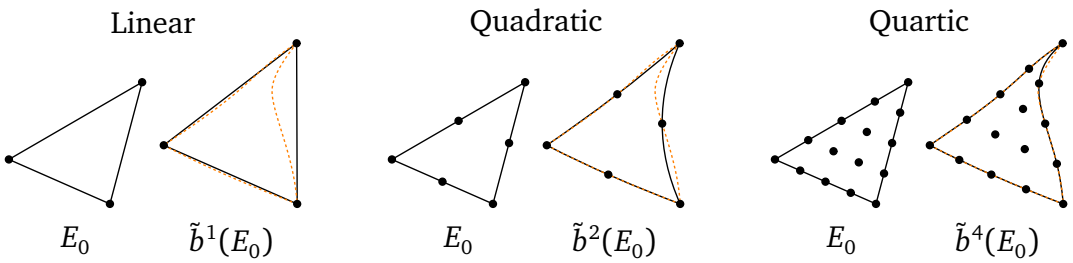


Figure 4.10. Comparison of b (orange dashed line) with its polynomial approximations $\tilde{b}^k$ (black solid line) for an element E_0.

matrix) and multiple time-steps (*e.g.,* in case of transient non-linear elasticity simulations). For instance, in Figure 4.1 the quantities related to b are computed only at the first time-step and reused in the following ones.

Despite the pre-computation, the evaluation of b remains expensive. Fortunately, mean value coordinates are straightforward to parallelize on shared memory processors. In fact, every point-wise evaluation of b and J_b can be computed in a completely independent way. Figure 4.9 shows the parallel-running times using OpenCL [74] with respect to different input sizes, computed on a laptop computer with Intel Core i7 2.3GHz processor and 16GB RAM.

4.3 Piecewise mapping approximations

As previously mentioned, composite mean-value mappings are computationally very expensive and their inverse is computed by solving an even more expensive optimization problem. An alternative option is approximating such mappings element-wise by a simpler geometric map. In this section we show possible choices of such approximation, ranging from polynomials (Section 4.3.1) to polygonal approximations (Section 4.3.2 and Section 4.3).

4.3.1 Polynomial elements

A natural choice of a piecewise approximation of b is polynomial mappings [43]. For each element E_0 of the source domain mesh $\mathcal{T}_0$ we consider the approximate geometric map

$$\tilde{b}^k(\boldsymbol{x}) = \sum_{j=1}^{m} c_j \varphi_j(\boldsymbol{x}), \tag{4.7}$$

where m is the number of interpolation nodes, φ_j is a polynomial of degree at most k, c_j is the associated coefficient, and $\boldsymbol{x} \in E_0$. The most basic procedure to

determine the coefficients c_j is to solve the following interpolation system [132]

$$\begin{bmatrix} \varphi_1(\boldsymbol{x}_1), & \varphi_2(\boldsymbol{x}_1), & \dots, & \varphi_m(\boldsymbol{x}_1) \\ \varphi_1(\boldsymbol{x}_2), & \varphi_2(\boldsymbol{x}_2), & \dots, & \varphi_m(\boldsymbol{x}_2) \\ \vdots & \vdots & & \vdots \\ \varphi_1(\boldsymbol{x}_m), & \varphi_2(\boldsymbol{x}_m), & \dots, & \varphi_m(\boldsymbol{x}_m) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} b(\boldsymbol{x}_1) \\ b(\boldsymbol{x}_2) \\ \vdots \\ b(\boldsymbol{x}_m) \end{bmatrix}.$$

Figure 4.10 shows different variants of $\tilde{b}^k$ for $k = 1, 2, 4$. Let

$$X_p^k(\mathscr{T}) := \{v \in C^0(\Omega) | \forall E \in \mathscr{T}\ \exists w \in \mathbb{P}_p : v(\boldsymbol{x}) = w((\tilde{b}^k G)^{-1}(\boldsymbol{x})), \forall \boldsymbol{x} \in E\} \quad (4.8)$$

be the finite element space associated with $\tilde{b}^k$, where $\mathbb{P}_p$ is the the space of polynomials of degree p in the reference element $\hat{E}$. When $p = k$ we are considering the case of iso-parametric finite elements, when $p < k$ we are considering super-parametric finite elements, and $p > k$ we are considering the sub-parametric case.

The main advantage of this approach is that, once the approximation $\tilde{b}^k$ is constructed, the original map b is not necessary anymore. In other words, the computation of b and its approximation procedure can be considered as preprocessing, which does not directly affect the performance of the solution process. Moreover, since $\tilde{b}^k$ is a polynomial, the numerical quadrature can be performed efficiently with floating point precision [127]. However, $\tilde{b}^k$ does not guarantee bijectivity. In fact, in the presence of large deformations or concavities this approach is very likely to fail. This issue is not resolved by naively refining the mesh as shown in Figure 4.13, but, in some cases, it can be alleviated by a suitable positioning of the interpolation points. In this thesis we do not address such issues and we consider exclusively standard node placements.

4.3.2 Polygonal elements

The main issue of the local polynomial approximant $\tilde{b}^k$ is caused by the limited (or absence of) control of the accuracy of the shape of the elements in the co-domain mesh $\mathscr{T}$. In other words, the shape of the element $\tilde{b}^k(E_0)$ solely depends on the choice of the interpolation points, which may result in the loss of bijectivity, as depicted in Figure 4.13.

An alternative method which reliably preserves bijectivity consists of approximating $b(E_0)$ by a polygon (or polyhedron in 3D) which resolution is controlled in the co-domain with arbitrary accuracy, see Figure 4.11 left. For obtaining such polygon, we first sample every side of E_0 with n uniformly sampled points

$\boldsymbol{x}_i$, $i = 1,\dots,n$. Then, we compute $b(\boldsymbol{x}_i)$ which gives us a densely sampled polygonal approximation of $b(E_0)$. Finally, for efficiency reasons, we discard all approximately collinear points thus creating polygons with fewer vertices. A point $\boldsymbol{x}_i$ is discarded if $u^T v/(\|u\|\|v\|) > (1-\varepsilon)$ is true, where $u = b(\boldsymbol{x}_i) - b(\boldsymbol{x}_{i-1})$, $v = b(\boldsymbol{x}_{i+1}) - b(\boldsymbol{x}_i)$, and $\varepsilon \in \mathbb{R}_{>0}$ determines the accuracy of the approximation. Since the original mapping b is rather local, our strategy naturally generates triangles away from the boundary, as shown in Figure 4.12. This brute force approach for approximating b can be replaced by adaptive discretization strategies with the primary objectives set to preserving bijectivity and the boundary shape. In this case adaptivity may substantially improve performance of the preprocessing phase and reduce the minimal number of degrees of freedom imposed to the solution process.

Standard finite element basis functions (*e.g.,* $\mathbb{P}_1$ and $\mathbb{P}_2$) are not suitable since the shape of the polygon is arbitrary. For this reason we follow the approach in [128] and employ *mean-value* basis functions ($\mathbb{MV}$), which are well defined for any polygon. Note that mean-value coordinates for triangles coincide with the $\mathbb{P}_1$ basis functions as for any other barycentric coordinate. When using such basis the assembly procedure is not performed in the reference element but directly in the physical element $E \in \mathscr{T}$. We describe this polygonal finite element space as

$$X_{\mathbb{MV}}(\mathscr{T}) := \{v \in C^0(\Omega) | \forall E \in \mathscr{T}\ \exists w \in \mathbb{MV}(E) : v(\boldsymbol{x}) = w(\boldsymbol{x}), \forall \boldsymbol{x} \in E\}, \quad (4.9)$$

where $\mathbb{MV}(E)$ are the mean value coordinates defined in the polygonal element E. Note that for this discretization we do not have an explicit geometric relationship (*i.e.,* volumetric map) between elements of $\mathscr{T}_0$ and the elements of $\mathscr{T}$ except for a node-wise correspondence on the boundary of each element. The main advantage of this discretization is that it avoids incurring in flipped triangles due to linear edges or self intersecting elements due to polynomial oscillations. Note that this discretization generally induces a higher number of degrees of freedom which are automatically determined in the proximity of the boundary.

4.3.3 Piecewise affine elements

A practical choice of geometric map between the polygonal approximations of element $E_0 \in \mathscr{T}_0$ and element $E \in \mathscr{T}$ (introduced in Section 4.3.2) is the piecewise affine map $\tilde{b}^A$. We construct $\tilde{b}^A$ by a means of suitable local triangulation which is valid for both polygonal elements E_0 and E. Validity of such map is ensured if the triangulation does not create degeneracies, such as flipped or zero area elements, in neither the polygonal approximations of E_0 nor E.

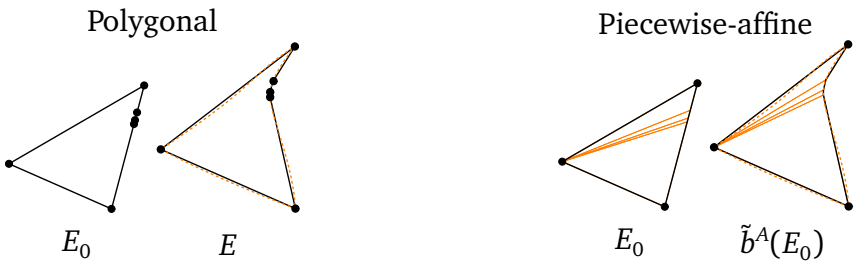


Figure 4.11. Comparison of b (orange dashed line) with its polygonal and piecewise-affine approximations (black solid line) for an element E_0.

The geometric map $\tilde{b}^A$ can be employed with different choices of finite element functions in the reference element, such as $\mathbb{P}_1$ and $\mathbb{P}_2$. We denote this finite element space as

$$X_p^A(\mathscr{T}) := \{v \in C^0(\Omega) | \forall E \in \mathscr{T}\ \exists w \in \mathbb{P}_p : v(\boldsymbol{x}) = w((\tilde{b}^A G)^{-1}(\boldsymbol{x})), \forall \boldsymbol{x} \in E\}, \tag{4.10}$$

where b^A is defined piecewise within each element as an affine transformation which maps each simplex $S_k^0 \subseteq E_0 \in \mathscr{T}_0$ to their image $S_k \subseteq E \in \mathscr{T}$. An example of element $\tilde{b}^A(E_0)$ is depicted in Figure 4.11 right.

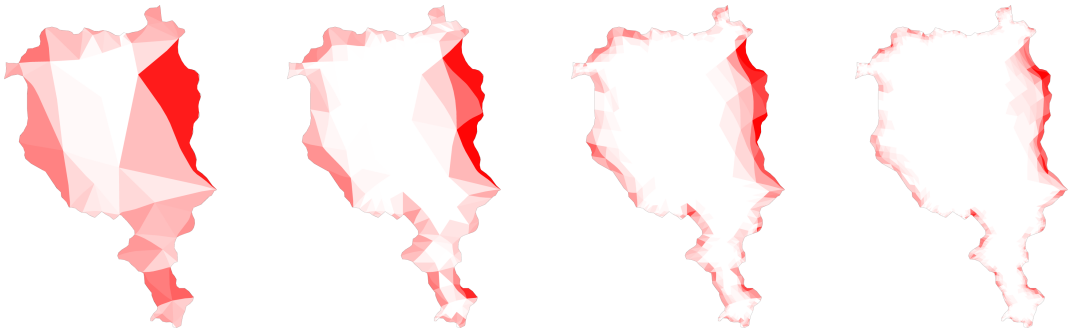


Figure 4.12. Number of nodes per element. The color represents the number of vertices of the polygonal elements, where white describes triangles and red more complex polygons.

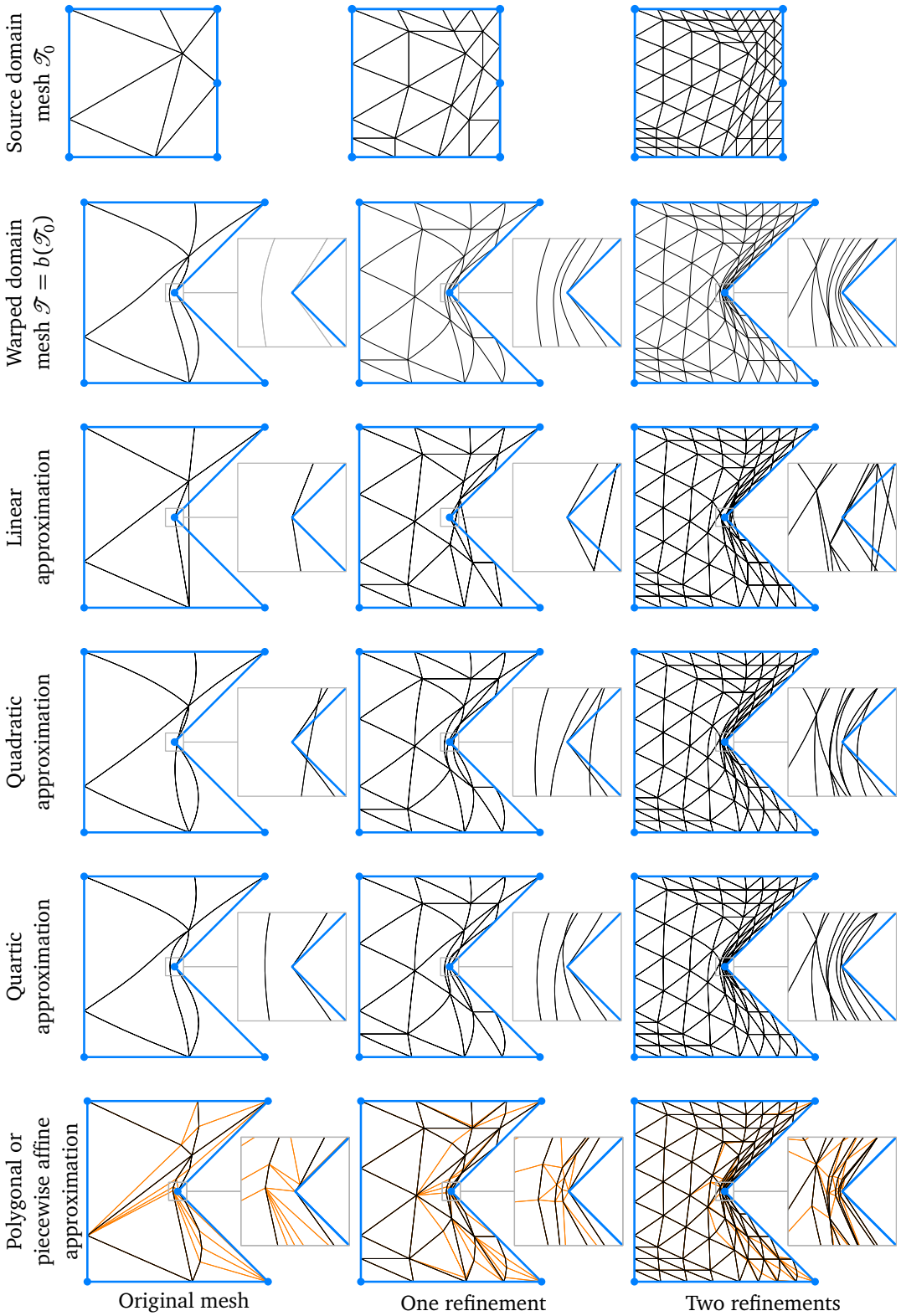


Figure 4.13. Comparison between b and its different element-wise approximations. The loss of bijectivity for linear, quadratic and quartic approximations around concavities is not mitigated by refinement. Note that refining the mesh may actually introduce this problem, as visible in the second column of the quartic approximation.

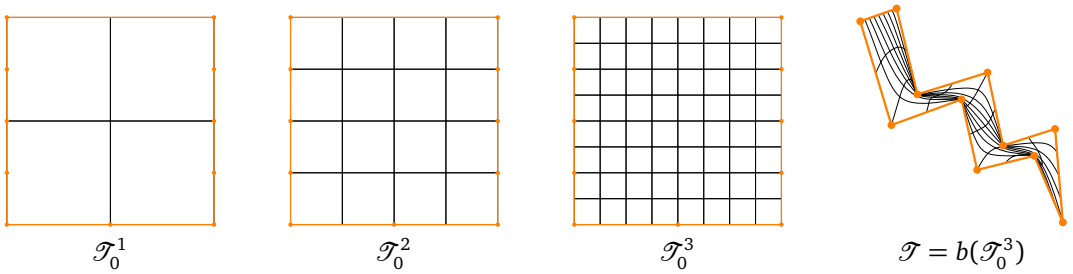


Figure 4.14. Multigrid method with warped mesh. Left: overview of the hierarchy of meshes. Right: warped mesh.

4.4 A multigrid method for arbitrarily shaped 2D meshes using parametric finite elements

In the context of geometric multigrid methods, employing parametric finite elements allows constructing the hierarchy of meshes in the parametrization domain, while the actual computational domain is represented exclusively through a geometric map. In this Section we briefly illustrate how to exploit the concepts introduced in Section 4.1 for implementing a geometric multigrid method for arbitrarily shaped meshes.

We consider the Poisson problem (4.1) and the discretization introduced in Section 4.1. We reuse the definition of the finite element space X_p^b, and introduce the following hierarchy of L nested spaces

$$\mathscr{H} = \{X_p^b(\mathscr{T}_0^1), \dots, X_p^b(\mathscr{T}_0^L)\},$$

where $\mathscr{T}_0^l$ is the conforming mesh at level l. The mesh at level l is the (uniform) refinement of the mesh at level $l-1$, in such a way that each element $E_0^l \in \mathscr{T}_0^l$ is a child of one element $E_0^{l-1} \in \mathscr{T}_0^{l-1}$, such that $E_0^l \subseteq E_0^{l-1}$ form a partition of E_0^{l-1}.

For relating coefficients between the different levels we employ the standard prolongation operator $\mathbf{I}^l : X_p^b(\mathscr{T}_0^{l-1}) \to X_p^b(\mathscr{T}_0^l)$ for geometric multigrid methods [21]. The standard procedure for constructing $\mathbf{I}^l$ in the case of nested spaces exclusively relies on the available hierarchical meta-information usually generated by the mesh refinement algorithm. The resulting operators can be equivalently constructed by assembling the pseudo-L^2-projection matrices (Section 2.2) between each adjacent level of the hierarchy of meshes in the parameterization domain [35; 62].

For constructing the mesh hierarchy in the parameterization domain we solely require $\mathscr{T}_0^1$, which is obtained by following the procedure described in Section 4.2, and construct the finer meshes by (uniform) refinement.

Once our hierarchy is constructed, we assemble the matrix $\mathbf{L}$ and related matrices and vectors from (4.4) on level L. We construct the stiffness matrices in the coarse levels by performing the Galerkin projections $\mathbf{L}^{l-1} = (\mathbf{I}^l)^T \mathbf{L}^l \mathbf{I}^l$. In a similar fashion, within the multigrid algorithm, we restrict the residual as $\mathbf{r}^{l-1} = (\mathbf{I}^l)^T \mathbf{r}^l$, where $\mathbf{r}^L = \mathbf{Mf} - \mathbf{Lu}$ and interpolate the correction as $\mathbf{c}^l = \mathbf{I}^l \mathbf{c}^{l-1}$.

Instead of employing the map b we can exploit its piecewise approximations for a more efficient finite element assembly. In the case of the piecewise polynomial approximation of b introduced in Section 4.3 we can just swap b with $\tilde{b}^k$ in the definitions of the multigrid hierarchy. With the finite element space $X_p^k(\mathscr{T}^l)$ from (4.7), where $\mathscr{T}_l = (\tilde{b}^k(\mathscr{T}_0^l))$, we describe the finite element space associated with level $l = 1, \ldots, L$ of the multigrid hierarchy

$$\mathscr{H}_p^k = \{X_p^k(\mathscr{T}^1), \ldots, X_p^k(\mathscr{T}^L)\}.$$

For the polygonal finite element approximation we have no explicit map approximation that we can employ. Hence, we define the coarse space $X_{\mathbb{MV}}^Q(\mathscr{T}^l) = \text{span}_{k \in J_l}\{\psi_k\}$ on the l level, where $J_l \subset \mathbb{N}$ is the index set of the nodes of $\mathscr{T}^l$. Let us introduce the weight matrix $Q \in \mathbb{R}^{|J_{l+1}| \times |J_l|}$ with elements q_{ij}, $\sum_{j \in J_l} q_{ij} = 1$, and follows the definition $\psi_i = \sum_{j \in J_l} q_{ij} \theta_j$, with $X_{\mathbb{MV}}^Q(\mathscr{T}^{l+1}) = \text{span}_{k \in J_{l+1}}\{\theta_k\}$. The weight matrix is constructed by assemblying the L^2-projection operator between the auxiliary spaces $X_{\mathbb{MV}}(\mathscr{T}_0^l)$, which are defined in the parameterization domain, as described in Section 2.2. Note that, in our set-up, for $l < L$ we have $X_{\mathbb{MV}}(\mathscr{T}_0^l) = X_1^1(\mathscr{T}_0^l)$. The recursive definition of $X_{\mathbb{MV}}^Q(\mathscr{T}^l)$ at its base case is defined as $X_{\mathbb{MV}}^Q(\mathscr{T}^L) := X_{\mathbb{MV}}(\mathscr{T})$, which allows us to express this multigrid hierarchy as

$$\mathscr{H}_{\mathbb{MV}} = \{X_{\mathbb{MV}}^Q(\mathscr{T}^1), \ldots, X_{\mathbb{MV}}^Q(\mathscr{T}^L)\}.$$

4.5 Chapter conclusion

The idea of combining the finite element method with bijective mappings allows representing complex geometries on coarse meshes and enables specifying interpolation conditions as in the classical finite element method. For instance, our method can be used with Lagrange elements, splines, NURBS, or mixed finite elements independently from the complexity of the input geometry. We introduce this novel discretization focusing on the particular case of composite mean value mappings which automatically creates a volume parameterization given only the boundary description.

Although we focus our presentation on the case of discrete geometry and composite mean value mappings, our construction might be suitable for any other

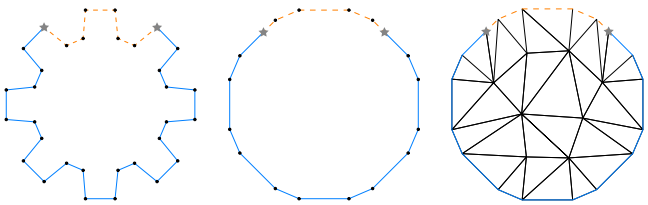


Figure 4.15. Handling the interface (grey stars) between the Neumann boundary (blue solid lines) and the the Dirichlet boundary (orange dashed lines) from Θ to $\mathscr{T}_0$.

choice of bijective mapping b, and it would be interesting to further investigate this flexibility. For instance, within the composite mapping, we can employ other types of smooth barycentric coordinates for which we can compute the Jacobian, such as maximum entropy coordinates [66; 52]. The method becomes computationally more expensive when employing the composite mean value mapping, however much of the related data can be precomputed and reused for different operators, as explained in Section 4.2.2. Moreover from the assembly point of view, our method only requires to change to quadrature procedure (4.5) by including the terms containing b.

In our presentation we first defined the mapping b as a global parameterization from Θ_0 to Θ, though, in order to have a faster computation of the quadrature data, we discussed strategies for localizing the mapping.

We presented the integration of our approach with efficient and modern solution techniques, such as multigrid methods. The flexibility provided by arbitrarily choosing the mesh for describing Ω_0, allows us to naturally generate nested geometric multigrid hierarchies with exact geometry. Moreover, the construction of the interpolation and restriction operators is trivially performed using standard mesh refinement of the source mesh $\mathscr{T}_0$, since the mapping b is the same for all levels.

Our discretization with composite mean value mapping enables treating boundary conditions with arbitrary precision even for the non-homogeneous case. For instance, let us consider the example problem in Figure 4.15, where Dirichlet boundary conditions are specified on $\partial\Omega_D \subseteq \partial\Omega$ (orange dashed lines) and Neumann conditions on $\partial\Omega_N = \partial\Omega \setminus \partial\Omega_D$ (blue solid lines). Let the interface (grey stars) between $\partial\Omega_D$ and $\partial\Omega_N$ be Γ and its corresponding interface in Θ_0 be Γ_0 (*i.e.,* $\Gamma = b(\Gamma_0)$). When generating the mesh $\mathscr{T}_0$ we preserve Γ_0 which is then mapped to its image Γ in $\mathscr{T}$. Since Γ is preserved, the boundary conditions which are specified on $\partial\Omega_D$ and $\partial\Omega_N$ can be equivalently handled on Ω_0.

A limitation which comes with our choice of bijective mapping b is that composite mean-value mappings do not provide any guarantees nor control over the quality of the computational mesh, which is usually determined by aspect-ratio and orientation of the elements [125]. In fact, the shape of the elements is entirely subject to any distortion caused by the mapping b. This suggests that further investigations on this topic are necessary for providing mesh-quality guarantees that are conforming to industry standards or enabling synergies with state-of-the-art methods, such as r-refinement [98]. In Section 4.3 we presented the element-wise approximations of b which may present future opportunities for localized mesh improvement and mesh adaption.

Chapter 5

Utopia: a C++ embedded domain specific language for scientific computing

In this chapter we present UTOPIA which is an eDSL deeply embedded in C++. Its philosophy is the separation of model and computation and its main purposes are linear and non-linear algebra, and finite element simulations. By exploiting meta-programming facilities, UTOPIA can easily be integrated with any other existing implementation, hence it is independent from technological changes. Moreover, UTOPIA shares the advantages of DSLs, for instance hidden parallelism, optimization transparency, and automatic differentiation. The UTOPIA eDSL is designed and developed for providing a balance between abstraction and low-level access without sacrificing performance. It aims at an organic integration with existing code without creating barriers between abstractions and implementation. In fact, both abstractions and low-level data are accessible to the user at any time. This allows users to extend their code with possibly missing functionalities by manipulating the low-level data (and back-end) directly. The flexible design of UTOPIA allows for adding these features in a straightforward way to future releases.

UTOPIA follows object oriented programming (OOP) principles [92]. An example is *design-by-contract* [96] which states that interfaces specified in the super-type have to be respected in the sub-type. A violation of the contract leads to code that, even if it compiles, does not run correctly. This type of fragility is common, and a way of handling this issue consists of hiding as many details as possible behind high-level abstractions. The UTOPIA abstraction performs all necessary work-around to ensure that the client code, when compiles and respects

the contract (which is automatically checked with assertions) runs without failures. Another important OOP principle is the *dependency inversion principle* [92] which states that high-level abstractions should dictate how high- and low-level modules have to be integrated. The implementation of UTOPIA is built and developed on top of this principle. Finally, other important OOP principles, such as the open/close principle, are taken into consideration when developing UTOPIA. The respect of these principles, allows UTOPIA to be modular, reusable, easy to extend, and sustainable to maintain.

This chapter is organized as follows: in Section 5.1 we explain the principles and design of UTOPIA, in Section 5.2 we present some extensions, and in Section 5.3 we provide a set of scenarios to show the usage of UTOPIA in an application environment.

5.1 Architecture

Many powerful linear algebra libraries (*e.g.,* PETSc, UBLAS [134], and ARMADILLO [119]) or finite element libraries (*e.g.,* LIBMESH, MFEM [77], DUNE [9], and FETK [65]) already exist. For this reason, the first prototype of UTOPIA does not "reinvent the wheel" and relies on PETSc for the algebra and LIBMESH for the finite element assembly. It is possible to develop other back-ends, such as automatic OPENCL code generators.

The design of the UTOPIA core is based on three main components: the *eDSL* (Section 5.1.1), the *expression-tree* (Section 5.1.2), and the *evaluator* (Section 5.1.3), which are represented in the component diagram in Figure 5.1. The UTOPIA eDSL allows users to state the behavior of their program and only care about the details relevant to their application. Despite this high degree of abstraction, users may need to perform operations on concrete data-types, such as accessing the entries of a matrix. To facilitate such tasks, UTOPIA provides an *application programming interface* (API) which directly queries the back-end for particular data (Section 5.1.4).

Natively, UTOPIA does not compute any of the specified operations since it relies solely on its back-ends. We use expression-trees to evaluate the operations depending either on the overall evaluation strategy or the specific back-end properties. We note that evaluation and back-end are conceptually tightly coupled. For instance, if the back-end is a library such as PETSc, we tailor the evaluation of the expression tree to the specific C functions.

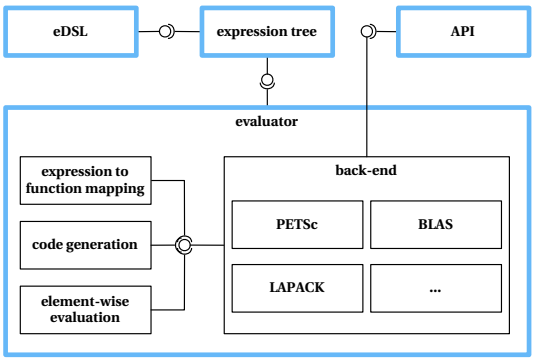


Figure 5.1. Component diagram of the UTOPIA core, the main components are highlighted. The connectors describe the dependency relationships among components, where the circle represents an interface and the arc represents the usage of such interface.

5.1.1 Embedded domain specific language

The UTOPIA eDSL primitives are mainly inspired by MATLAB and EIGEN [55], and are realized by exploiting the C++ language meta-programming facilities (*i.e.,* templates), function overloading, and functional-style programming constructs introduced in the C++11 standard. The simple and clean presentation to users is made possible by type inference and the `auto` keyword which allows to hide complex meta types, see Figure 5.2.

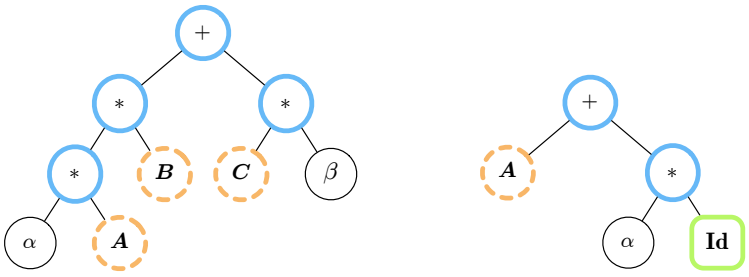
Tensor types are represented by the *wrapper* class `template<class Tensor, int Order> class Wrapper` within UTOPIA expression-trees. The first template parameter `Tensor` is the concrete back-end type, for instance a PETSc matrix. The second parameter `Order` is the tensorial order, for instance `Wrapper<Tensor, 1>` describes vectors and `Wrapper<Tensor, 2>` describes matrices. Interaction between wrappers is defined through the UTOPIA primitives, for instance the multiplication or transpose operator. This interaction automatically generates an expression tree which is evaluated only when it is assigned to another wrapper object. Direct wrapper manipulation (*e.g.,* changing the entries of a vector) is implemented in a unified API (Section 5.1.4).

5.1.2 Expression tree

The nodes of the tree are expressions and can be either operations, wrapper objects, or factories. The actual expression tree is generated by combining multiple expression nodes, as shown in Figure 5.3. Operations are branches of the tree

```
// 1) types
Matrix A, B, C;
double alpha, beta;

// 2) complete type of the expression treee
Binary< Binary< Number<double>,
                Multiply<Matrix,
                         Matrix>,
                Multiplies>,
        Binary< Number<double>,
                Matrix>,
        Plus> expr = alpha * A * B + beta + C;

// 3) using C++11 auto keyword
auto expr = alpha * A * B + beta + C;

// 4) the expression is evaluated here
Matrix value = expr;
```

Figure 5.2. Four block of code showing the C++ representation of UTOPIA expressions. The second block of code shows the type of the expression tree generated from expression $\alpha\mathbf{AB} + \beta\mathbf{C}$. The third block shows the usage of the auto keyword, at this stage no computation involved. The last block shows how to trigger the evaluation.



$\alpha\mathbf{AB} + \beta\mathbf{C}$ $\mathbf{A} + \alpha\mathbf{Id}$

Figure 5.3. Expression tree of different expressions: the solid blue circles represent operations, the dashed orange circles are wrappers, and the solid green boxes describe factories.

and represent specific operations between nodes of the tree. For instance, the expression $x + y$ is translated to a binary node representing the addition between its left and right subtrees. Wrappers are always leaves of the expression tree and allow to determine the back-end and the evaluation strategy. This is realized by propagating meta-information from the leaves to the root at compile time. Factories are also leaves and are implicit data-descriptions. Their primary objective is the creation of objects such as the zero, identity, and sparse matrices. The secondary objective is specifying compositions of expressions without actually creating concrete tensor objects. For instance, the expression $\mathbf{A}+\mathbf{Id}$ is translated to an addition binary node whose right child is the identity factory, which is then implemented as a diagonal shift in our PETSc back-end.

Every function call or operation on the tree is exclusively performed on the type `template<class Derived> Expression` (in short `Expression<Derived>`), which conforms to the *curiously recurring template pattern* (CRTP), where `Derived` is subclass of `Expression`. CRTP allows for static polymorphism which comes with three main advantages. First, everything can be recognized by its most specific type at any point in the code. This allows treating different types in a specific way within the same code by function overloading and template specialization. Second, static polymorphism allows for more complex type based transformations, such as symbolic differentiation at compile time. Third, the eDSL baseline performance. This is possible since dynamic polymorphism is not necessary (no virtual table search is performed), and the compiler has all necessary information for in-lining. In fact, the overall performance solely depends on back-end libraries and their integration with the eEDSL.

In order to generate such tree, UTOPIA provides a well defined set of primitives, which are fully described in the complete API documentation [142]. For instance, Figure 5.4 shows that the absolute value operation returns a unary expression of type `Unary<Expr, Operation>` , where the absolute value function is applied to the `Expr` expression-tree, and `Operation` is the `Abs` type. Another example is the addition operation defined as the operator `+`. This operation returns a binary expression of type `Binary<Left, Right, Operation>`, where `Left` and `Right` are two expression-tree operands and `Operation` is the `Plus` type. Note that nodes do not have behavior and they do not store any data.

An expression tree is used in two ways: for directly computing the corresponding numerical operations, or for applying specific transformations beforehand, such as symbolic differentiation or optimizations. Its first usage is the evaluation of the represented expressions. The evaluation is triggered when using the assignment operator with a concrete type as left operand. For instance the expression $\mathbf{b} = \mathbf{Ax}$ is translated to an object of type `Assign<Vector,`

`Multiply<Matrix, Vector> >` which is forwarded to the `Evaluator` (Section 5.1.3).

The second usage, is transforming the tree to another one with different properties. For instance, it can be transformed for optimization purposes by means of tree simplification or re-ordering, or for symbolic differentiation. Since there is no actual computation needed in the transformation of the expression tree any manipulation is completely independent from the actual implementation in the back-end.

An example of performance optimization involves the composite operation **ABCx**, where **A**, **B**, **C** are matrices, and **x** a vector. Reordering the operations from (**ABC**)**x**, to **A**(**B**(**Cx**)) decreases the time complexity of the operation from cubic to quadratic. Reordering allows reducing the number of expression-tree types that are generated by transforming equivalent composite operations to one common type (*e.g.,* $\alpha\mathbf{x}+\mathbf{y}$ and $\mathbf{y}+\alpha\mathbf{x}$), thus limiting the amount of code required for implementing a back-end. Note that reordering is performed while generating the tree and after the complete tree is available. The only exception is trivial reorderings for expressions such as $\alpha\mathbf{x}$ and $\mathbf{x}\alpha$, which is performed before generating the expression tree.

An example of symbolic matrix differentiation [103] is the computation of the derivative of $\mathbf{x}^T\mathbf{A}\mathbf{x}+\mathbf{x}^T\mathbf{b}$ with respect to **x**. This is realized by transforming this expression with compile time decisions to $\mathbf{A}\mathbf{x}+\mathbf{b}$ without any actual numerical computation.

The adoption of statically typed expression-trees allows propagating meta and structural information from the leaves to the root at compile time, hence without any runtime cost. This allows making informed decisions, possibly at compile time, on how to approach the tree evaluation. The statically propagated information includes sparsity (*e.g.,* dense, sparse, diagonal, scalar *etc.*) and back-end types. This information can be used for back-end specific optimization in the tree evaluation, or for compile time checks of available features. In fact, this

```
template<class Expr>
Unary<Expr, Abs> abs(const Expression<Expr> &expr) {
  return expr.derived();
}

template<class Left, class Right>
Binary<Left, Right, Plus> operator+(const Expression<Left>  &left,
                                    const Expression<Right> &right) {
  return Binary<Left, Right, Plus>(left.derived(), right.derived());
}
```

Figure 5.4. Implementation of the eDSL primitive for the absolute value and the addition.

allows writing code that, once compiled, runs without encountering unsupported operations, broken interfaces, or errors.

5.1.3 Evaluator

The purpose of the evaluator component is computing tensorial quantities from a given expression tree by means of back-ends (Section 5.1.3). This is performed with three different strategies: *expression to function mapping* (Section 5.1.3), where the evaluator works as a dispatcher forwarding calls directly to specific back-end functions; *code generation* (Section 5.1.3), where the evaluator uses the expression-tree as a guide for generating and compiling code on the fly; *element-wise evaluation* (Section 5.1.3), where the evaluator evaluates the expression-tree, completely or partially, in-line [133]. The design of the evaluator provides facilities for simple and modular extensions of the delegation of function calls, allowing to map composite expressions to the most specific code without changing the interface to the user.

We note that the aforementioned strategies can form a synergy. By exploiting statically typed expression-trees, we can match particular branches to specific strategies. For instance, when evaluating expression templates by in-line operation, the lack of specificity in the evaluation of an expression (*e.g.,* matrix matrix multiplication) might dramatically affect performance in a negative way (due to cache misses). However, creating many intermediate representation might be inefficient as well, since memory allocation is very costly. The combination of different strategies can overcome this problems.

Back-end

The back-end provides data-types and algorithms. Usually it is either an external library, such as PETSc or uBLAS, or a composition of libraries. In order to conform to a common interface, libraries might be wrapped into an interface adapter. The evaluator binds the eDSL abstractions to the concrete types and algorithms of specific libraries depending on the desired strategy. The back-end adapter is also used by the API functions for accessing structural information such as matrix sizes or entries.

Expression to function mapping

This strategy is performed by mapping expressions to functions of specific back-ends. This is done by matching the types of specific expression-trees (or sub-

```
void build(int n, double val, std::vector<double> &v) {
  v.resize(n);
  std::fill(v.begin(), v.end(), val);
}
```

Figure 5.5. Example back-end implementation for the factory function `values`.

trees) by means of (partial) template specialization and delegating the computation to specific functions. The functions are applied in a functional programming style deemed possible by C++11 *return value optimization* (RVO) and move semantics. For instance, the factory `Vector v = values(n, 0.1);` in our custom back-end is mapped to the function in Figure 5.5 which constructs a vector of length `n` with entries equal to 0.1.

More complex composite expressions (*i.e.,* sub-trees) are mapped to specific back-end calls; for instance the vectorial expression $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ is mapped to the BLAS function `axpy`. Another example is matricial expression $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$ which is mapped to the function `dgemm_` in BLAS for which a representation of the tree is depicted in Figure 5.3. Similar mappings are possible in the PETSc back-end, for instance the evaluation of the a triple matrix product of the form $\mathbf{P}^T\mathbf{AP}$ is mapped to the PETSc function `MatPtAP`, as shown in Figure 5.6.

The minimal requirements for a back-end are to map basic operations, such as addition and multiplication. The mapping of composite expressions to specific function calls can be gradually integrated. As a consequence, all the existing application codes will automatically benefit from the performance given by the specific function without changing a line of code.

Code generation

This strategy aims to generate code in a different language such as OpenCL. In such a way the evaluator generates, compiles, and runs programs following a just-in-time (JIT) approach similarly to ViennaCL. Statically typed expressions allow automatically generating and compiling specific sub-trees only once for each runtime. A given expression-tree is divided into several sub-trees which represent the concurrent portions of the algorithm where synchronization is not required. For each of these sub-trees we generate and compile a computational kernel, or retrieve an already compiled one. The implementation of this particular part of Utopia is in a primitive stage and it only serves as a proof-of-concept.

```
// empty eval declaration which is specialized for all expressions
template<class Expr, // the pattern to match
         class Traits = utopia::Traits<Expr>,
         int Backend  = Traits::Backend>
class Eval {};

// specialization for the triple product transpose(L) * A * R
template<class LAndR, class A, class Traits>
class Eval<Multiply< Multiply<Transposed<LAndR>, A>, LAndR>,
           Traits,
           PETSC> {
public:
  typedef utopia::Multiply< Multiply<Transposed<LAndR>, A>, LAndR> Expr;
  typedef EXPR_TYPE(Traits, Expr) Result;

  static Result apply(const Expr &expr) {
    Result result;

    // check if leftmost and rightmost operands are the same object
    if(&expr.left().left().expr() == &expr.right()) {
      // access back-end singleton and perform optimal triple product
      UTOPIA_BACKEND(Traits).triple_product_PtAP(
        Eval<LAndR, Traits>::apply(expr.left().left().expr()),
        Eval<A, Traits>::apply(expr.right()),
        result
      );
    } else {
      // perform general triple product L^T A R
    }

    return result;
  }
};
```

Figure 5.6. Mapping the triple matrix product $\mathbf{P}^T\mathbf{AP}$ to the PETSc function `MatPtAP`.

Element-wise evaluation

Element-wise evaluation exploits the wrapper API for evaluating an expression tree following the typical implementation of *expression-templates meta-programming*. Expression templates are a well known and widely used technique for linear algebra libraries, such as EIGEN, and have two main advantages. First, element-wise operations can be concatenated and evaluated without creating intermediate data. This potentially allows the compiler to "in-line" operations [69] and achieve comparable performance with respect to the most specific code for a particular task. Second, it allows to write expressions using operators thus providing an aesthetic syntax similar to the classical mathematical writing.

In UTOPIA, this type of evaluation also allows for convenient interoperability between wrappers belonging to different back-ends without requiring neither copies nor conversions. Additionally, instead of directly evaluating expressions through the API, we can exploit libraries such as EIGEN as back-ends by translating the UTOPIA expression-tree (Section 5.1.2) directly to back-end representation.

5.1.4 API and memory access transparency

The eDSL is accompanied by a uniform API for basic interactions with tensors, such as accessors and mutators (or getters and setters). Nevertheless, back-end types are accessible by means of the `raw_type` function which takes an UTOPIA tensor and returns its back-end representation. This allows users to directly manipulate the back-end representation, for instance to add specific missing features.

Although UTOPIA provides a certain degree of transparency and strives for simplicity, it requires that operations are handled in a "memory-conscious" manner, as shown in Figure 5.7. Since UTOPIA targets large scale computations and heterogeneous computing, three main aspects related to memory location need to be explicitly handled.

The first aspect concerns distributed memory access. For instance, compute nodes of a supercomputer have separated dedicated memory, which implies that the data accessible to one node is not directly accessible to another one. In fact, independently of the particular back-end, with UTOPIA it is mandatory to use ranges and their associated functions to deal with data distributions. Range objects allow to iterate over elements of tensors, which are available in the local address space.

The second aspect regards data acquisition. For instance, a memory region on

```
// n x n sparse matrix
SizeType n = 100;
SizeType max_entries_x_row = 3;
SparseMatrix m = sparse(n, n, max_entries_x_row);

{ // beginning of write lock scope
  Write<SparseMatrix> w(m);
  Range r = row_range(m);

  for(SizeType i = r.begin(); i != r.end(); ++i) {
    if(i > 0) {
      m.add(i, i - 1, -1.0);
    }

    if(i < n-1) {
      m.add(i, i + 1, -1.0);
    }

    m.add(i, i, 2.0);
  }
} // end of write lock scope
```

Figure 5.7. Assembly of 1D Laplacian on template class `SparseMatrix`.

a GPU device is not directly accessible by the CPU, hence it needs to be copied to main memory to be read. To handle different address spaces, UTOPIA provide a locking mechanism of resources. In fact, in order to read or write from and object we need to acquire its lock and release it when we are done. When we use a `Read` lock, the memory is copied from the device to the main memory, whereas, when we use a `Write` lock, memory is copied from a temporary buffer to the device memory. This mechanism is automatic and the data-transfer is performed when a lock is created (for reading) or destroyed (for writing).

The third aspect covers ownership of ordered data. When writing in a distributed matrix, the physical memory location of the entries might not be directly accessible. To hide this problem from the user, UTOPIA uses locks again, and, when the write lock is destroyed, all non-local data is automatically communicated at once based on their global index.

The locking mechanism can be abused to perform post-processing. For instance changing the matrix internal representation to better fit the sparsity pattern.

5.2 Extensions

On top of the algebraic primitives, UTOPIA provides a simple interface to linear and non-linear solution strategies (Section 5.9). Since UTOPIA is an eDSL tar-

geting scientific computing it includes a prototype for finite element assembly (Section 5.2.2). Finally, for facilitating debugging activities, UTOPIA is accompanied by two visualization tools, one allowing to inspect algebraic data, and the other to display functions on 3D meshes (Section 5.2.3).

5.2.1 Solvers as eDSL primitives

In UTOPIA, the representations of direct and iterative solution methods are designed for optimization problems arising from partial differential equations. They conform to the same idea applied to algebraic expressions, that is the separation of model and computation. This design methodology follows two directions. On the one hand, we reuse as many existing implementations as possible by interfacing with external solvers, such as PETSc's KSP and SNES. For instance, the `LUFactorization<Matrix, Vector>` class uses different implementations such as LAPACK for our custom back-end, or MUMPS [2] for PETSc. This might not apply to all solvers, since some implementations might be unavailable for a particular configuration, which results in the application not compiling.

On the other hand, we develop new generic solvers on top of UTOPIA algebraic primitives, hence they can be used with any wrapper. For instance, our implementation of the trust-region algorithm is the same for our custom back-end and code-generation back-end. The only difference is how the primitive operations are performed.

In order to exploit a wide range of existing scientific software applications UTOPIA ensures interoperability with finite element libraries such as FENICS and MOOSE. The class `Function` (Figure 5.8) is designed to ensure this interoperability, by providing a uniform interface between external libraries and UTOPIA solvers.

The interface to UTOPIA solvers is designed with several levels of abstraction. Users may call the high level routine `solve()`. This routine does not expose any detail and uses a default strategy to solve a system of equations. However, some problems require different solution strategies. For instance, for symmetric-positive-definite systems we can use the conjugate-gradient method, while for non-symmetric and indefinite systems we can use the preconditioned-generalized-minimal-residual (GMRES).

The modular design of solvers allows compositions of different strategies. This enables users to combine and customize different solution strategies in order to build an efficient solver which suits their application the best, see Example 2.

```
template<class Matrix, class Vector>
class NonlinearFunction : public Function<Matrix, Vector> {
public:
  typedef UTOPIA_SCALAR(Matrix) Scalar;

  bool value(const Vector &x, Scalar &f) const override {
    // evaluation routine for objective function
    return true;
  }

  bool gradient(const Vector &x, Vector &g) const override {
    // evaluation routine for gradient
    return true;
  }

  bool hessian(const Vector &x, Matrix &H) const override {
    // evaluation routine for Hessian
    return true;
  }
};

NonlinearFunction<Matrix, Vector> fun;
Vector x = values(2, 0);
solve(fun, x);
```

Figure 5.8. Function used with UTOPIA non-linear solvers.

5.2.2 Finite element assembly

The set of UTOPIA primitives can be easily extended to include other domain specific applications. In this section we describe the finite elements primivites of UTOPIA eEDSL. This extension provides a set of primitives for describing multi-linear forms arising from variational problems. A typical example is: find $\mathbf{u} \in \mathbf{V}_h \subset H^1$ such that

$$a(\mathbf{u}, \mathbf{v}) = f(\mathbf{v}) \quad \forall \mathbf{v} \in \mathbf{W}_h,$$

where $a : \mathbf{V}_h \times \mathbf{W}_h \to \mathbb{R}$ is a bilinear form, $f : \mathbf{W}_h \to \mathbb{R}$ is a linear form, $\mathbf{V}_h$ and $\mathbf{W}_h$ are (tensor) finite elements spaces, H^1 is a Hilbert space of weakly differentiable funcions, and h is the discretization paramter. A standard choice is low order Lagrange elements (*e.g.,* $\mathbb{P}_1$), hower higher orders can be chosen freely.

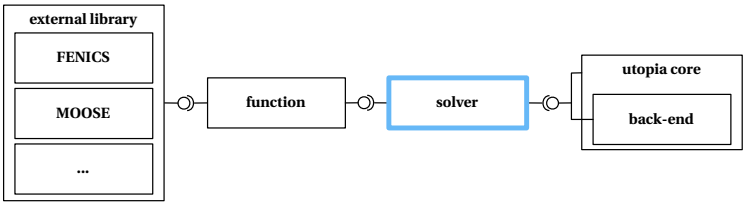


Figure 5.9. Component diagram of UTOPIA solvers.

In addition to the basic tensorial representation introduced in Section 5.1.1, new types are necessary for representing finite element spaces, their basis functions, and related coefficients. The lexicon of UTOPIA is extended with new functionalities specific to the finite element representations and assembly. For example, for instantiating a function `v` from its function space `V` it is sufficient to write `v = fe_function(V)`. Constant and non-constant coefficients can be instantiated by means of `coeff`, `vec_coeff` for vector valued coefficients, and `mat_coeff` for matrix valued coefficient. All of these objects can be manipulated with differential operators such as `grad`, `div`, `curl`, and `integral`.

The evaluator is developed with the specific goal of finite-element based assembly, since most of the objects are tensorial functions (*e.g.,* basis functions). In fact, all quantities are evaluated at quadrature points, which means that operations are applied to collection of tensorial values.

Note that, UTOPIA's finite element eDSL does not prescribe that the assembly procedure has to be either global or element-wise (local). Hence, the assembly can be implemented at any level in the code, not only at the top level. This flexibility allows to straightforwardly employ the eDSL together with elaborate solvers. For instance, varations of the multigrid algorithm may require separate element matrices, for instance to generate coarse representations of the operator by means of spectral agglomeration [28]. We refer to Section 5.3 for examples created with our prototype built with a LIBMESH back-end.

5.2.3 Visualization and debugging

Debugging is a difficult task, and as Reiss wrote [113], "we need to make using software visualization for debugging the standard practice of all programmers". This task is even more challenging in the context of numerical simulations because of the complexity and size of the underlying data. Several solutions are available such as Paraview [5], VisIt [25], and Vestige [122]. These tools also provide support for "in-situ visualization" [116], which couples the visualization with the simulation code such that the data is visualized while the simulation is running. On the one hand, we integrated UTOPIA with the Vestige visualization tool. On the other hand, we developed a specific visualization for the distributed and transient algebraic data.

The visualization of the algebraic data follows the same philosophy as proposed in [122]: the application code sends the data through a socket to an application running on an independent process which provides both visualization and inspection facilities. This separation allows to follow the evolution of the algebraic data at several moments of an algorithm at the same time.

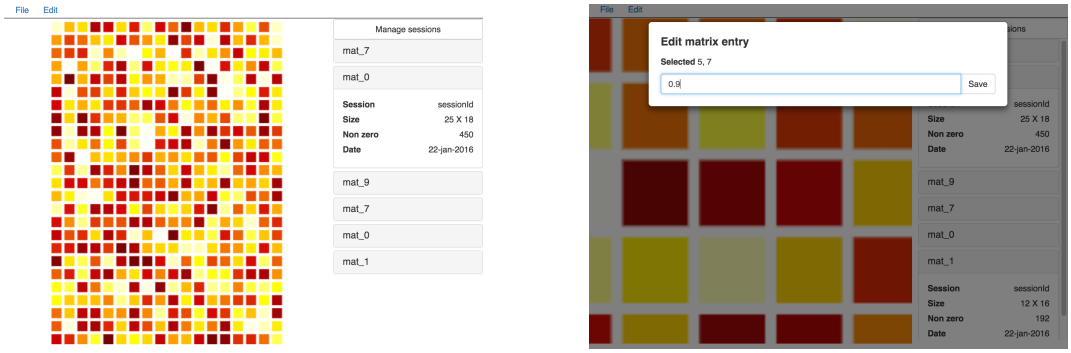


Figure 5.10. Visualization of algebraic data with our companion tool.

Figure 5.10 shows an example visualization with our tool, where the entries of a matrix are color-coded according to their value. The tool stores the data of several sessions and visualizes them in the list on the right, which displays also high-level information, such as matrix size and non-zeros entries. In contrast with [122] and other similar software tools, our tool also allows manipulating the objects, as it is visible in the right picture.

5.3 Applications

In this section we provide several illustrative examples showing the usage of UTOPIA. To simplify the explanation we first introduce the necessary notation. Let $\Omega \subset \mathbb{R}^d$ be a (bounded) domain with Lipschitz boundary $\Gamma = \partial\Omega$, and let $L^2(\Omega)$ be the Hilbert space of square integrable functions on Ω with inner product

$$(v, w) = (v, w)_{L^2(\Omega)} = \int_\Omega vw \, d\mathbf{x}$$

and norm $\|\cdot\| = \|\cdot\|_{L^2(\Omega)} = (\cdot,\cdot)^{1/2}_{L^2(\Omega)}$. With Γ_D we denote the Dirichlet boundary and with $\Gamma_N = \Gamma \setminus \Gamma_D$ the Neumann boundary.

Let $V = V(\Omega)$ be the function space associated with Ω, $\mathbf{V} = V^d$ the d-th order product space of vector-valued functions, $\mathbf{W} = V^{d\times d}$ the respective space of matrix-valued functions. Naturally, these spaces are discretized by means of finite elements. In our examples, we employ Lagrange elements such as $\mathbb{P}_1$ for linear simplicial elements, and $\mathbb{Q}_1$ for bilinear/trilinear elements.

Example 1 shows how a text-book pseudo code translates to the UTOPIA eDSL. Note that the translation is one-to-one and it preserves the same level of simplicity, while being completely parallel. Example 2 shows the combination of

UTOPIA solvers with the MOOSE library. Interoperability is achieved by means of the `SNESAdapter`. This interface uses the PETSc SNES data structure to pass information about objective functions and their derivatives to the non-linear solution method. Consequently, it can be easily used with other FEM libraries built on top of PETSc, such as the MOOSE framework [24].

Example 3 shows how to use our eDSL for specifying a non-linear anisotropic-Poisson problem with solution-dependent diffusion coefficients. In this example the coefficient-function `f` is specified using the C++11 lambda function `rhs_fun`. Example 4 shows, how to specify an initial value problem,

$$\partial u(x,t)/\partial t = h(t,u(x,t),\ldots)$$

with $u(x,t_0) = u_0(x)$, by exploiting the UTOPIA primitive `dt` for identifying the time derivative. From the variational formulation we automatically extract the update function h which is used in a time integrator, such as the explicit Euler or the Runge-Kutta.

The last two examples show mixed formulations which can be interpreted as the variational problem: find $u \in V, \boldsymbol{\sigma} \in \mathbf{W}$:

$$\begin{aligned} a(u,v) + b(\boldsymbol{\sigma},v) &= k(v) \\ b(\mathbf{q},u) + g(\boldsymbol{\sigma},\mathbf{q}) &= d(\mathbf{q}) \qquad \forall v \in V, \mathbf{q} \in \mathbf{W}, \end{aligned}$$

where a, b, g are bilinear-forms, and k, d are linear-forms. This problem is is mirrored in the code by the corresponding block linear system:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{G} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\sigma} \end{bmatrix} = \begin{bmatrix} \mathbf{k} \\ \mathbf{d} \end{bmatrix}.$$

Example 5 shows how to assemble such mixed finite element problem derived from a least-squares functional [102] which ensures the ellipticity of the resulting linear operator. Similarly, Example 6 shows how to assemble a least-squares linear-elasticity problem. Note that in this example, the UTOPIA eDSL is used in a larger environment which deals with contact problems [33].

Example 1: Preconditioned conjugate gradient

Description

Algorithm taken from [126].

Pseudo-code and parallel code

```
                                    Vector r, d, q, s;
                                    double delta_new, delta_0, delta_old;
                                    double alpha, beta;

i ← 0                               int i = 0;
r ← b − Ax                          r = b - A * x;
d ← M^{-1} r                        solve(M, r, d);
δ_new ← r^T d                       delta_new = dot(r, d);
δ_0 ← δ_new                         delta_0 = delta_new;
While i < i_max ∧ δ_new > ε^2 δ_0   while(i < i_max && delta_new > eps_2 * delta_0) {
  q ← Ad                              q = A * d;
  α ← δ_new / (d^T q)                 alpha = delta_new/dot(d, q);
  x ← x + αd                          x += alpha * d;

  If i is divisible by 50             if(i % 50 == 0)
    r ← b − Ax                          r = b - A * x;
  Else                                else
    r ← r − αq                          r -= alpha * q;

  s ← M^{-1} r                        solve(M, r, s);
  δ_old ← δ_new                       delta_old = delta_new;
  δ_new ← r^T s                       delta_new = dot(r, s);
  β ← δ_new / δ_old                   beta = delta_new/delta_old;
  d ← s + βd                          d = s + beta * d;
  i ← i + 1                           ++i;
                                    }
```

Example 2: Phase field fracture

Description

Find $\mathbf{u} \in \mathbf{V}_t,\ c \in M_t$, and $\beta \in Q_t$, such that

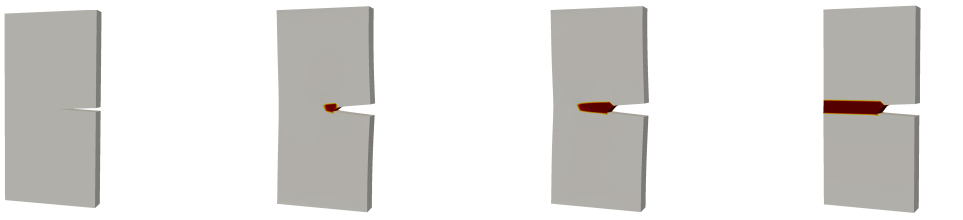
$$\begin{aligned}\left([(1-c)^2+k]\boldsymbol{\sigma}_0^+ - \boldsymbol{\sigma}_0^-, \nabla \mathbf{v}\right) &= (\mathbf{t}, \mathbf{v})_{\Gamma_N} \\ (l\nabla c, \nabla q) + (\beta, q) &= (l\nabla c \cdot \mathbf{n}, q)_{\Gamma_N} \\ \left(\frac{\partial c}{\partial t} - \frac{1}{\eta}\left\langle \beta + 2(1-c)\frac{\psi_0^+}{g_c} - \frac{c}{l}\right\rangle_+, m\right) &= 0,\end{aligned}$$

$\forall \mathbf{v} \in \mathbf{V}_t,\ \forall q \in Q_t,\ \forall m \in M_t$, where $t \in [0,T],\ c \in [0,1],\ g_c, l, \eta,\ k \in \mathbb{R}$, $\langle x \rangle_+ = (|x| + x)/2$ is the ramp function, and the $\boldsymbol{\sigma}^+$ and $\boldsymbol{\sigma}^-$ super-scripts respectively represent the positive and negative parts of the stress; see [97]. We perform our experiment in a parallelepipedal domain, and we discretize $\mathbf{V}_t$ with $\mathbb{P}_1^d$ elements, and both M_t and Q_t with $\mathbb{P}_1$ elements. The computation is performed in a displacement-driven context where the rate of change $\mathbf{u}$ on the y-axis is 10^{-5} on the top side and zero on the bottom side. We use a linear elastic material with Lamé parameters $\lambda = 12$, $\mu = 8$, and phase-field parameters $\eta = 5 \times 10^{-4}$, $l = 0.022$, and $k = 10^{-8}$.

Code

```
auto direct_solver = make_shared<LUDecomposition<Matrix, Vector>>();
auto smoother      = make_shared<GaussSeidel<Matrix, Vector>>();

// initialization of preconditioner, setting up interpolation operators
Multigrid<Matrix, Vector> mg(smoother, direct_solver);
mg.init(move(interpolation_operators));
mg.set_max_iter(1); mg.set_cycle_type(2);

// iterative linear solver with MG as a preconditioner
ConjugateGradient<Matrix, Vector> cg; cg.set_preconditioner(make_ref(mg));

// non-linear solver
Newton<Matrix, Vector> solver(cg);
solver.set_line_search_strategy(make_shared<Backtracking<Matrix, Vector>>());

// interface between MOOSE and utopia non-linear solvers
SNESAdapter<Matrix, Vector> fun(snes); solver.solve(fun, x);
```

Simulation



$u_y = 0$ $u_y = 5.9 \times 10^{-3}$ $u_y = 6.3 \times 10^{-3}$ $u_y = 7.4 \times 10^{-3}$

Crack pattern for different values of u_y. The color represents the solution for the phase-field parameter c, from the unbroken state $c = 0$ (gray) to the fully broken state $c = 1$ (red).

Example 3: Non-linear anisotropic Poisson problem

Description

Find $u \in V$:

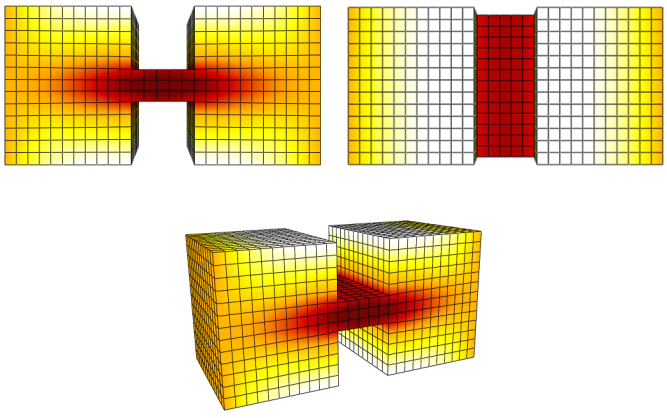
$$(1/(u^2 + 0.1)\mathbf{A}\nabla u, \nabla v) = (f, v),$$

$\forall v \in V$, where $f \in V$, $u|_{\partial\Omega} = g$, and $\mathbf{A} \in \mathbb{R}^{d\times d}$.
We perform our simulation with two connected cubes, with paramters $f = 10\|\mathbf{x}\|_2 - 5$ for $\mathbf{x} \in \Omega$, $g = 0$, $\mathbf{A} = \text{diag}(10, 0.1, 1)$, and we discretize V with $\mathbb{Q}_1$ elements.

Code

```
int dim = 2;
// anisotropic diffusion tensor
Matrix A = identity(dim, dim);
A.set(0, 0, 10);
A.set(1, 1, 0.1);
A.set(2, 2, 1);

// right-hand side
std::function<void(const Point &p, Scalar &ret)>
rhs_fun = [dim](const Point &p, Scalar &ret) -> void {
  // right-hand side function code...
};

auto f = coeff(rhs_fun);

// solution
auto u_k = interpolate( coeff(1.0), Vh, make_ref(solution_vec) );

// bilinear form
auto bf = integral( dot(1./(pow2(u_k) + coeff(0.1)) * A * grad(u), grad(v)) );

// linear form
auto lf = integral(dot(f, v));
```

Simulation



Visualization of the solution u from different persectives.

Example 4: Heat-equation
Description

Find $u \in V$:

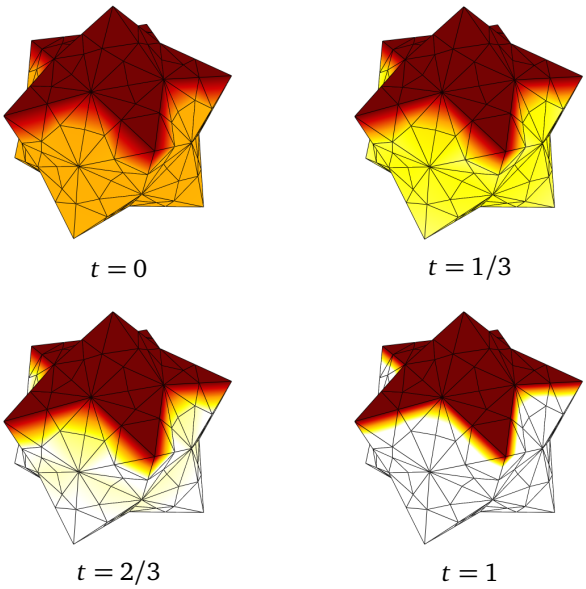
$$\left(\frac{\partial u}{\partial t}, v\right) = (f, v) - c(\nabla u, \nabla v),$$

$\forall v \in V$, where $u(\mathbf{x}, t_0) = u_0$, $u|_{\Gamma_d} = g$, $f \in V$ and $c \in \mathbb{R}$; see [108].
We perform our simulation in a star-shaped domain with $u_0 = 1$, $c = 2$, $g = 0$, and Γ_D as the top surface of the star. We discretize V with $\mathbb{P}_1$ elements.

Code

```
std::function<void(const Point &p, Scalar &ret)>
rhs_fun = [dim](const Point &p, Scalar &ret) -> void {
  // right-hand side function code...
};

double c = 1.0;
auto f = coeff(rhs_fun);

// u is created and set to u_0 = 1
auto u  = interpolate(coeff(1.0), Vh, make_ref(solution_vec));
auto eq = integral(dot(dt(u), v)) == integral(dot(f, v) - c * dot(grad(u), grad(v)));

explicit_euler_integrate(eq, t_start, dt, t_end, [](const double t) {
  // intercept each time-step of the simulation and perform custom operations
  // the current solution is stored in u
});
```

Simulation



Visualization of the solution u at different time-steps t.

Example 5: Least-squares Helmholtz equation
Description

Find $u \in V$, $\boldsymbol{\sigma} \in \mathbf{W}$:

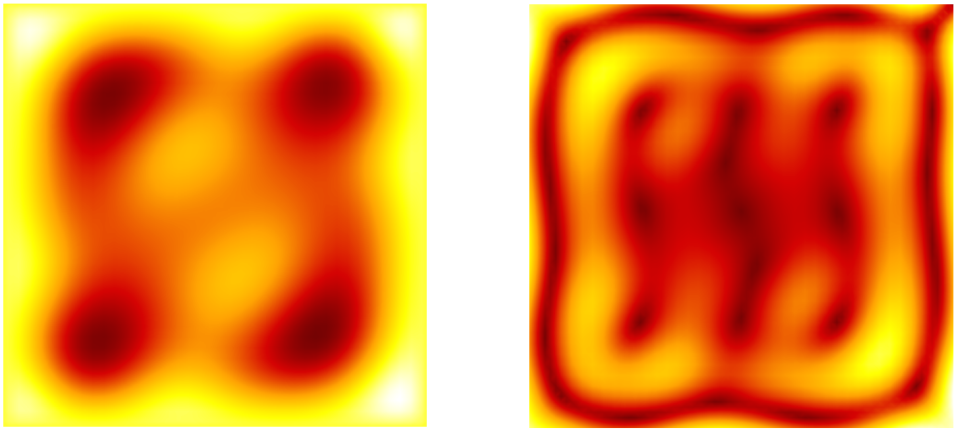
$$(c\,u, c\,v) + (\nabla u, \nabla v) + (\operatorname{div}\boldsymbol{\sigma}, c\,v) + (\boldsymbol{\sigma}, \nabla v) = (f, c\,v)$$
$$(c\,u, \operatorname{div}\mathbf{q}) + (\nabla u, \mathbf{q}) + (\boldsymbol{\sigma}, \mathbf{q}) + (\operatorname{div}\boldsymbol{\sigma}, \operatorname{div}\mathbf{q}) + \beta(\operatorname{curl}\boldsymbol{\sigma}, \operatorname{curl}\mathbf{q}) = (f, \operatorname{div}\mathbf{q}),$$

$\forall v \in V, \forall \mathbf{q} \in \mathbf{W}$, with $f \in V$, $u|_{\Gamma_D} = g, c \in \mathbb{R}_{<0}$, and $\beta \in \mathbb{R}_{>0}$; see [102].
We perform out simulation in a square domain with $g = 0$, $f = 1$, $c = -100$, $\beta = 0.99$, $\Gamma_D = \Gamma$, and we discretize V with $\mathbb{Q}_1$ elements and $\mathbf{W}$ with $\mathbb{Q}_1^d$ elements.

Code

```
// parameters
double c    = -100.0;
double beta = 0.99;
auto f      = coeff(1);

// bilinear forms
auto bf_11 = integral((c*c) * dot(u, u) + dot(grad(u), grad(u)));
auto bf_12 = integral(c * dot(div(s), u) + dot(s, grad(u)));
auto bf_21 = integral(c * dot(u, div(s)) + dot(grad(u), s));
auto bf_22 = integral(dot(s, s) + dot(div(s), div(s)) +
                      beta * dot(curl(s), curl(s)));

// linear forms
auto lf_1 = integral(c * dot(f, u));
auto lf_2 = integral(dot(f, div(s)));
```

Simulation



u $\|\boldsymbol{\sigma}\|_2$

Visualization of the solutions u and $\boldsymbol{\sigma}$.

Example 6: Linear elasticity with least-squares finite elements

Description

Find $\mathbf{u} \in \mathbf{V}, \boldsymbol{\sigma} \in \mathbf{W}$

$$(\varepsilon(\mathbf{u}), \varepsilon(\mathbf{v})) - (\mathscr{A}\boldsymbol{\sigma}, \varepsilon(\mathbf{v})) = \mathbf{0}$$
$$-(\varepsilon(\mathbf{u}), \mathscr{A}\mathbf{q}) + (\operatorname{div}\boldsymbol{\sigma}, \operatorname{div}\mathbf{q}) + (\mathscr{A}\boldsymbol{\sigma}, \mathscr{A}\mathbf{q}) = -(\mathbf{f}, \operatorname{div}\mathbf{q}),$$

$\forall \mathbf{v} \in \mathbf{V}$ and $\forall \boldsymbol{q} \in \mathbf{W}$, where $\mathbf{f} \in \mathbf{V}$, $\mathbf{u}|_{\Gamma_D} = \mathbf{g}$, $\boldsymbol{\sigma}|_{\Gamma_D} = \mathbf{z}$, $\varepsilon(\mathbf{w}) = (\nabla\mathbf{w} + (\nabla\mathbf{w})^T)/2$, and

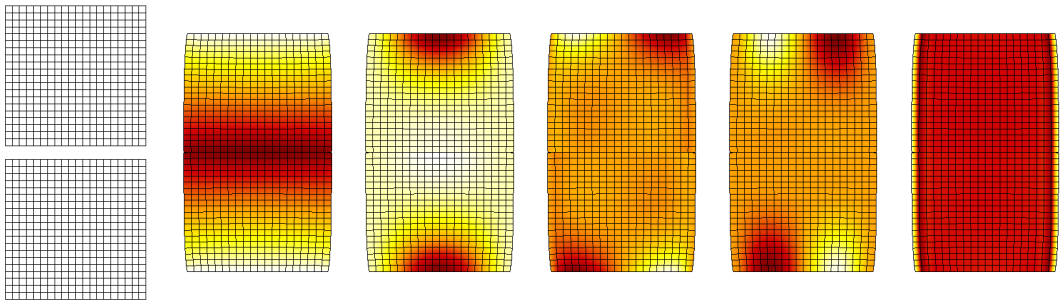
$$\mathscr{A} : \mathbb{R}^{d\times d} \to \mathbb{R}^{d\times d}, \qquad \mathscr{A}\boldsymbol{\sigma} = \frac{1}{2\mu}(\boldsymbol{\sigma} - \frac{\lambda}{d\lambda + 2\mu}(\operatorname{tr}\boldsymbol{\sigma})\mathbf{I})$$

is the inverse strain-stress relationship tensor and $\lambda, \mu \in \mathbb{R}$ are the Lamé paramters; see [23]. We simulate the contact betweend two unit sized squares with $\lambda = 1, \mu = 1$, and $\mathbf{f} = \mathbf{0}$. We specify the boundary conditions $\mathbf{g} = -[0.2, 0.2]$ on the top side, $\mathbf{g} = [0.2, 0.2]$ on the bottom, $\boldsymbol{z} = \mathbf{0}$ on the left and right sides. Contact conditions are resolved according as explained in Section 2.3.1. We discretize $\mathbf{V}$ with $\mathbb{Q}_1^2$ elements and $\mathbf{W}$ with $\mathbb{Q}_1^{2\times 2}$ elements.

Code

```
auto A   = stress_strain_rel_tensor(dim, mu, lambda);
auto f   = vec_coeff(force);

auto e   = 0.5 * (transpose(grad(u)) + grad(u));
auto As  = A * s;

// bilinear forms
auto b11 = integral(inner(e, e));
auto b12 = integral(-inner(As, e));
auto b21 = integral(-inner(e, As));
auto b22 = integral(inner(div(s), div(s)) + inner(As, As));

// linear forms
auto l1  = integral(inner(vec_coeff(0., 0.), u));
auto l2  = integral(-inner(f, div(s)));
```

Simulation



Input $\|\mathbf{u}\|_2$ $\boldsymbol{\sigma}_{11}$ $\boldsymbol{\sigma}_{12}$ $\boldsymbol{\sigma}_{21}$ $\boldsymbol{\sigma}_{22}$

Viusalization of the solutions $\mathbf{u}$ and $\boldsymbol{\sigma}$.

5.4 Chapter conclusion

UTOPIA is a unified C++ eDSL for non-linear algebra and finite element assembly following the philosophy of separation of model and computation. With a MATLAB-like *look-and-feel*, our eDSL supports existing state-of-the-art software libraries, code generation and expression templates. UTOPIA has lower barriers to entry to parallel computing, since it provides purposefully partial parallelization, data-distribution and memory-location transparency. Moreover, UTOPIA provides simple debugging routines to visualize both numerical and structural data. Since solution methods, both linear and non-linear, are a fundamental building block of any scientific software our eDSL is extended to provide a coherent interface for supporting a large variety of strategies. These features are shown in several examples of solution strategies and variational formulations in finite element assembly.

One current major performance issue in our back-end implementations is memory allocation. Though this issue can be addressed specifically in each back-end, we plan to develop an independent mechanism to automatically solve this problem. Our idea is to provide memory pools which reduce the amount of allocations by reusing intermediate representations. Another performance improvement consists of providing primitives to handle sparsity explicitly, for instance allocating a sparse matrix with the same sparsity pattern as another one (*e.g.,* `m2 = sparse(sparsity(m1))`). Finally, a minor issue of UTOPIA is the compilation time due to the heavy use of template types. This issue will be attenuated by exploiting C++ modules which are currently being standardized for future version of C++ [38].

A relevant MATLAB primitive is index-sets. This primitive allows accessing tensorial entries based on a set of indices. Unfortunately, this feature is not yet available in UTOPIA hence it will be added in the near future. We plan to extend the eDSL with primitives to allow transfer of discrete fields [81] for non-conforming domain decomposition methods [13; 137; 105]. Another interesting language feature is variational inequalities which allows specifying obstacle or contact problems [34]. The UTOPIA eDSL already contains a basic prototype of symbolic differentiation [39; 56], we aim to improve it and add it to the finite element eDSL. When the symbolic version does not apply, we intend to include automatic differentiation mechanisms [4].

Finally, we plan to integrate and develop new back-ends for both the algebra and the finite element assembly. This will allow to benchmark the different libraries within the same framework.

Chapter 6

Numerical experiments

In this chapter we look into the runtime performance of our parallel information transfer algorithm (Section 6.1), and we observe the numerical behavior of solving the Poisson problem discretized with our parametric finite element discretization (Section 6.2).

6.1 Parallel transfer

The first contribution of the parallel approach presented in this Chapter 3 is to enable for really complex and difficult simulation scenarios to be handled. Nevertheless, in this section we illustrate scaling studies in the weak-scaling and strong-scaling settings. We also illustrate particular corner cases where the approach does not perform at its best, and provide considerations on output sensitivity and its effect on scaling. Being a method for handling an output-sensitive problem, the issue of scaling should neither be addressed nor observed as in standard and ideal scenarios. We measure scaling/speed-up by $s = t_B/t_P$, where t_P is the time of the run with P processes and t_B is the time of the base run with B processes.

In our experiment we measure the cost of the assembly of the transfer operator starting when the input mesh is received, hence the measurement includes searching, computing intersections, generating quadrature formulas, computing the local integrals, and delivering the two coupling operators in their sparse matrix representation.

As approximation spaces we have chosen linear Lagrangian finite-element spaces. As a measure of the output we count the number of intersections which is equivalent to the number of evaluated integrals. The cost of integration may vary with respect to the shape of the intersection.

In order to provide an estimate of the cost of the assembly of the transfer operator in comparison to a standard mass matrix assembly we performed an experiment in serial with the same routines that we use in our parallel numerical experiments. The finite element assembly is performed in a generic software framework which allows for mixed formulations with customizable quadrature formulas, and our measurements are performed for a meshed cube $\mathscr{T}$ with 297 316 elements. For assembling the transfer operator we use mass-matrix assembly calls with Petrov-Galerkin formulation (different trial and test spaces) and special quadrature formulas which are generated as explained in Section 2.3. We consider the computational time of the assembly procedure a standard mass matrix and compare it with the computational time for computing the pseudo-L^2 projection operator. This particular transfer operator is constructed for transferring between equal spaces both associated with the same mesh, hence resulting in an identity matrix. The measurements include the computational time associated with the intersection detection, intersection computation, quadrature points generation, and assembly. The observed ratio between the assembly time of the transfer operator and the assembly time of the mass matrix is approximately 15, and the larger portion of the cost is due to intersection computation and the assembly. The measurements of our experiments are tagged and organized as follows:

- *Create adapters*: the cost of creating the adapter representations from the provided geometric data. An adapter allows representing an element and its related mesh data in a suitable format for the library code abstractions. In order to this these adapters include meta-information such as tags, domain markers, and geometric information such as AABBs and k-DOPs.

- *Build tree/detection*: the cost of constructing the octree, searching for matching remote nodes, generating index-sets for handling both the nodes and the geometric data.

- *Load-balancing*: the cost of linearizing the local trees and scheduling the narrow-phase detection.

- *Organize dependencies*: organization and communication of the actual geometric data.

- *Match and re-balance*: cost of the narrow-phase detection, and re-balancing. Here no actual intersection is computed, only bounding-volume matching is performed.

- *Computation*: intersection computation and assembly.

We can consider all the measurements except the computation, as the overhead which results in using our algorithm.

6.1.1 Hardware

The studies have been performed at the Swiss Supercomputing Center (CSCS) on a Cray XC40 with the following specification: 1256 Compute Nodes with 2 Intel® Xeon® E5-2690 v3 @ 2.60GHz (12 cores each, 24 virtual cores each with hyperthreading enabled); Theoretical Peak Performance 1.254 Petaflops; Memory Capacity per node 64 GB (1192 nodes) and 128 GB (64 fat nodes, bigmem); Memory Bandwidth per node up to 137 GB/s per node; Total System Memory 82.5 TB DDR3; Peak Network Bisection Bandwidth of 4.5 TB/s; Parallel File System Peak Performance of 50 GB/s;

6.1.2 Weak-scaling experiments

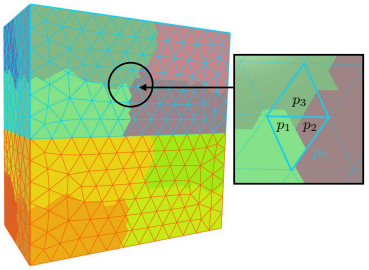
With weak scaling, we investigate how the framework behaves, with respect to computational time, when increasing the number of processors, and keeping the amount of computation per process fixed. The problem is output-sensitive, which means that the computational complexity depends on the size of the output, making it difficult, for most scenarios, to study scaling in a fair way by just controlling the size of the input.

Hence, we study weak scaling in the simplest scenario, depicted in Figure 6.1. We have a stack of parallelepipeds, each parallelepiped has two resolutions a fine mesh and a coarse mesh. The partitions of coarse and fine meshes are randomly distributed also with respect to each other. Hence, intersecting elements of the fine mesh and coarse mesh are likely to be owned by different processes, thus stored in different memory address spaces. In this setting, we assemble the transfer operator for transferring from the coarse space to the fine space.

In Figure 6.1, we see the scaling results for this experiment in two resolutions. In Figure 6.2, we have a detailed illustration of the medium size experiment.

6.1.3 Strong-scaling experiments

With strong scaling, we investigate how the framework behaves with respect to computational time, when increasing the number of processes, and keeping the total size of the problem fixed. The charts in Figure 6.3 and Figure 6.4, illustrate



Stacked decomposed multi-grid parallelepipeds. Color represents processes.

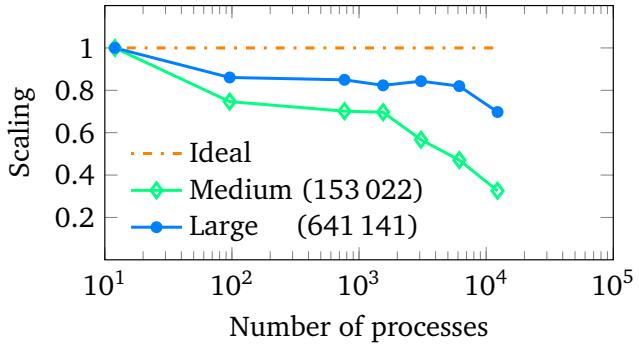
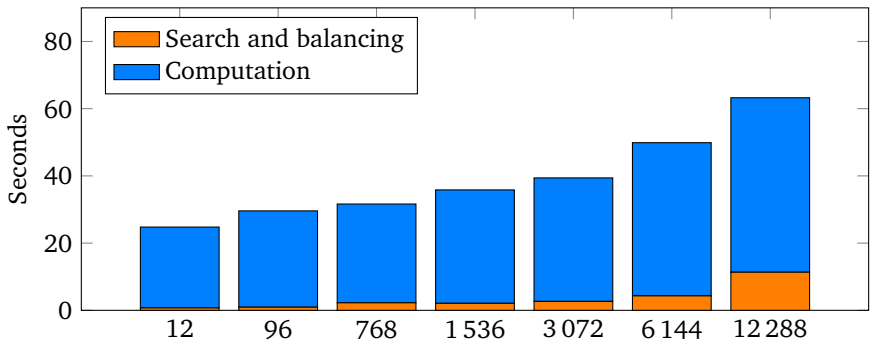


Figure 6.1. Weak scaling with different resolutions. Medium: per process 10 923 input, 43 691 output. Large: per process 153 022 input, 641 141 output. See Figure 6.2 for more details about the medium size experiment.



Create adapters	0.05	0.05	0.05	0.05	0.05	0.05	0.05
Build tree/detection	0.26	0.44	1.68	1.54	2.10	3.59	10.78
Load-balancing	0.00	0.00	0.00	0.00	0.00	0.01	0.00
Organize dependencies	0.19	0.21	0.24	0.24	0.24	0.30	0.31
Match and re-balance	0.20	0.20	0.24	0.24	0.24	0.31	0.25
Computation	24.07	28.66	29.39	33.73	36.74	45.58	63.25
Total	24.78	29.58	31.63	35.83	39.40	49.87	74.60
Processes	**12**	**96**	**768**	**1536**	**3 072**	**6 144**	**12 288**
Tree-depth	5	5	5	6	6	6	6
Input size ($\log_2$)	17	20	23	24	25	26	27
Output size ($\log_2$)	19	22	25	26	27	28	29

Figure 6.2. Volume projections: weak scaling experiment. The x axis describe the number of processes. The computational times is measured in seconds. The input is about 10 200 tetrahedral elements per process. *Search and balancing* includes all the measurements except the *computation*.

Create adapters	0.337	0.254	0.201	0.149	0.116
Build tree/detection	0.672	0.321	0.247	0.133	0.087
Load-balancing	0.004	0.002	0.001	0.001	0.001
Organize dependencies	1.119	1.363	1.420	1.451	1.679
Match and re-balance	1.699	1.058	0.853	0.634	0.449
Computation	11.000	5.090	2.469	1.248	0.584
Total	14.830	8.090	5.194	3.618	2.919
Processes	**288**	**576**	**864**	**1 536**	**3 072**
Tree-depth	5	5	5	5	5

Table 6.1. Surface projections: strong scaling experiment. Time in seconds for the middle size experiment illustrated in Figure 6.4.

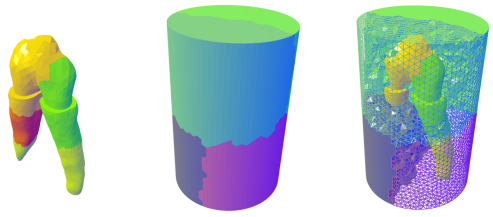
the scaling for experiments with different mesh resolutions. Table 6.1, illustrates in detail the computational time of each phase for different number of processes, of the experiment shown in Figure 6.4(f) and (g).

6.1.4 Particular scenarios

This approach can handle any random spatial distribution, however, in the worst case scenario (*e.g.,* elements are distributed completely at random) where we have an almost all-to-all dependency graph, no significant advantage is taken from parallel tree-search algorithm in terms of scaling.

User input and parameter tuning

For surface projections, the user input can help to improve the performance dramatically, since the search is bounded to a particular distance. In fact, the user can specify a parameter ϵ which determines the size of the bounding volume of each element, by blowing it up in normal direction. The value of ϵ affects the search and the quantity of element-pairs detected as near. In the experiment depicted in Figure 6.5 the bounding volumes are larger (hence large ϵ) than needed which gives rise to many false positives. In order to have an idea of how this affects performance, we ran the software twice on eight cores, and we observed that when reducing ϵ by 40% we decreased the number of false positives (of about 60%), and saved 60% of the computational time.

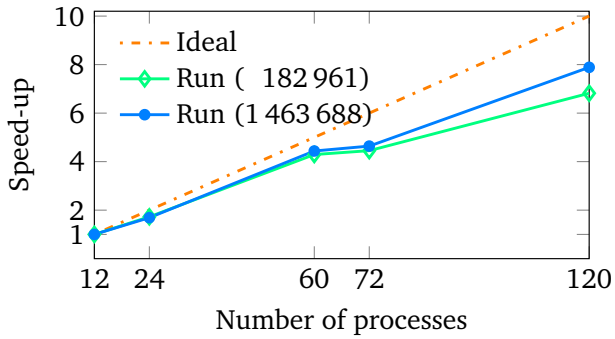


(a) A tooth and a cylinder.

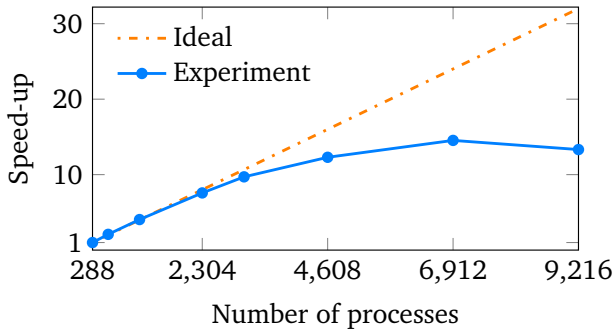
(b) Set-up.

Input size	**182 961**	**1 463 688**
Output size	399 499	3 344 987
Candidates	821 262	7 013 103
False positives	421 763	3 668 116
Base experiment (seconds)	19.9	172.5

(c) Statistics for experiment (d).



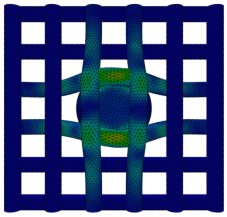
(d) Two small size output experiments.



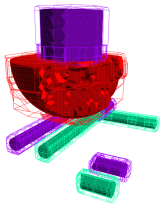
Input size	**93 676 032**
Output size	239 453 927
Candidates	526 199 291
False positives	286 745 364
Base experiment (seconds)	742.5

(e) Medium size output experiment.

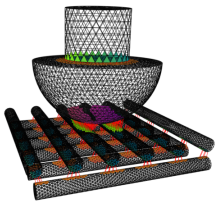
Figure 6.3. Volume projections: strong scaling experiment for different resolutions. In (a), (b) we see the set-up of the three experiments; here color represents processes. In (c) and (d) we see two small size experiments, and in (e) one greater size experiment.



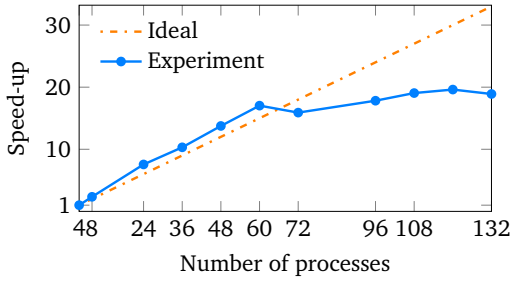
(a) Example multi-body contact simulation.



(b) Bounding volume hierarchy constructed by the algorithm.

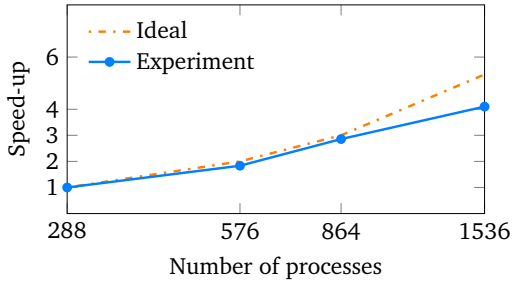


(c) Detected contact boundary and gaps.



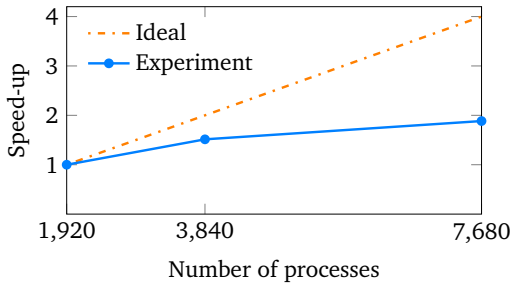
Volume elements	11 608 960
Surface adapters	529 648
Output size	73 207
Candidates	1 249 892
False positives	1 176 685
Base experiment (seconds)	9.6

(d) Small.



Volume elements	742 973 440
Surface adapters	8 474 368
Output size	929 688
Candidates	156 505 837
False positives	155 576 149
Base experiment (seconds)	14.8

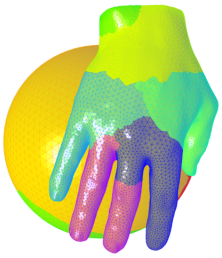
(e) Medium.



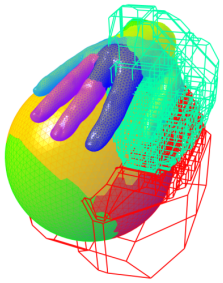
Volume elements	5 943 787 520
Surface adapters	33 897 472
Output size	3 464 544
Candidates	2 163 911 881
False positives	2 160 447 333
Base experiment (seconds)	35.3

(f) Large.

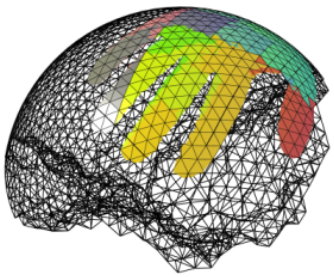
Figure 6.4. Surface projections: strong scaling experiment with different resolutions. In (a), (b), and (c) is depicted the context of the experiment. The coloring: in (a) it is the Von-Mises stress, in (b) it represent the process. The scaling results exclusively include the cost of computing the transfer operator related quantities. In experiment (d), above 60 processes we can see the search costs taking over, and the total time stagnates at around 0.5 seconds. Similarly, in experiment (f), the search occupies the 70% of the total time.



Surface proximity detection problem.



Bounding volume hierarchy.



Detail of the geometric surface projection.

Figure 6.5. Predicting the contact region: when there is *a-priori* knowledge about the problem, the scope of the search can be reduced for saving computational time. Color represent processes.

6.1.5 Scaling and output-sensitivity

The main reason for the observed scaling behavior, in both strong and weak scaling studies, is mainly due to imbalance and synchronization waiting time in the search phase. The imbalance is due to the initial geometric set-up, when the meshes are distributed in an unbalanced way as for instance in the scenario presented in Section 6.1.3, or the actual output of the search is strongly unbalanced. In fact, since we are treating an output-sensitive problem, the actual cost of the search is unknown *a-priori* and depends directly on the output, *i.e.,* the number of candidate intersections, which is directly related to the spatial location of the meshes. Once the intersection candidates are found, and we have the necessary knowledge, the assembly procedure can be performed in a more balanced way. However, also the actual assembly might be subject to unavoidable imbalance depending on the actual computed intersection polytopes and number of quadrature points which are generated on each process. A possible solution might be to re-balance again after the computation of the intersections.

6.2 Parametric finite elements with bijective mappings

We focus our study on (mostly) super-parametric discretizations based on composite mean value mappings (Section 2.5.1) and its approximations (Section 4.3) with linear Lagrange elements ($\mathbb{P}_1$). For our experiments the analytical solution is unknown, hence we estimate it by computing a reference solution $u \in X_1^1(\mathscr{T}_f)$ on a very fine mesh $\mathscr{T}_f$. To evaluate the quality of our discretization and the standard discretization, we compute different solutions u_h for several mesh sizes

notation	functions	geometric map	section	equation
X_1^b	$\mathbb{P}_1$	composite mean value	4.1	(4.3)
X_1^1	$\mathbb{P}_1$	affine	4.3.1	(4.8)
X_1^2	$\mathbb{P}_1$	quadratic	4.3.1	(4.8)
X_1^3	$\mathbb{P}_1$	cubic	4.3.1	(4.8)
$X_{\mathbb{MV}}$	$\mathbb{MV}$	–	4.3.3	(4.9)
X_1^A	$\mathbb{P}_1$	piecewise affine	4.3.3	(4.10)

Table 6.2. Finite element spaces employed in our experiments and where to find their definitions.

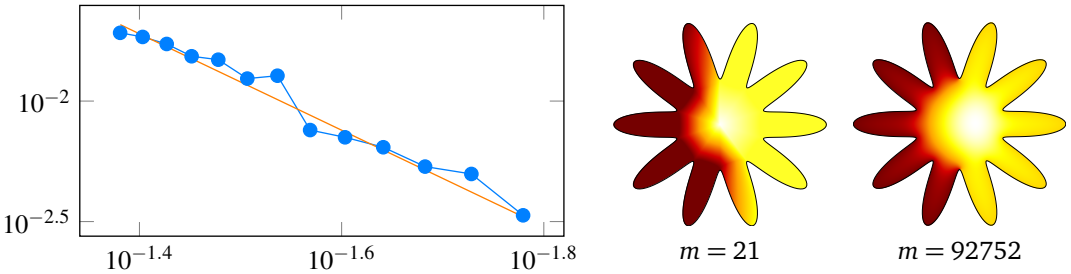


Figure 6.6. Left: visualization of $e(u_h)$ against the mesh size h, where the straight line shows the quadratic trend. Right: solution of the Poisson problem for different number of nodes m.

h. Table 6.2 provides an overview of the different spaces and notation appearing in this section.

6.2.1 Convergence

We exclusively study the convergence of the solution for $u_h \in X_1^b$ since it is the only discretization we introduced that provides an exact geometric description of the computational domain Ω. The solution is expected to converge quadratically in $L^2(\Omega)$ to the exact one with respect to the mesh size h for classical FEM with linear elements for H^2-regular problems. Hence, we study the convergence related to our approach by measuring the approximation error as

$$e(u_h) = \|\mathscr{P}(u_h) - u\|_{L^2(\mathscr{T}_f)},$$

where $\mathscr{P} : X_1^b(\mathscr{T}) \to X_1^1(\mathscr{T}_f)$ is the L^2-projection operator [137; 81] (the assembly of $\mathscr{P}$ by considering only the parameterization domain). Similar to standard

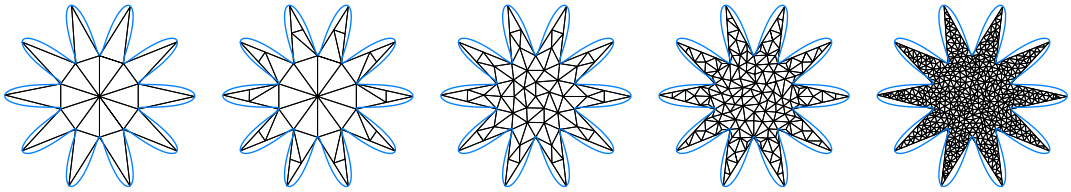


Figure 6.7. Mesh refinement without shape recovery. Even at fine resolution (last image) we do not recover the original shape (blue polygon).

FEM, our method shows a quadratic convergence behaviour for the Poisson problem, as illustrated in the plot in Figure 6.6. Despite the fact that the computation is always performed in the exact geometry, the approximation error is not zero because of the piecewise polynomial approximation of the solution, which is visible for a mesh with small m and disappears for larger m.

6.2.2 Comparison

We compare our discretization with the standard finite element discretization for a simple 2D problem (Figure 6.8), an extreme 2D problem (Figure 6.9), and for a realistic 3D shape (Figure 6.10). Since for the standard finite element discretization, the boundary of $\mathscr{T}$ differs from Ω, we measure

$$r(u_h) = \left| \frac{\|u_h\|_{L^2(\mathscr{T})}}{\|u\|_{L^2(\mathscr{T}_f)}} - 1 \right|$$

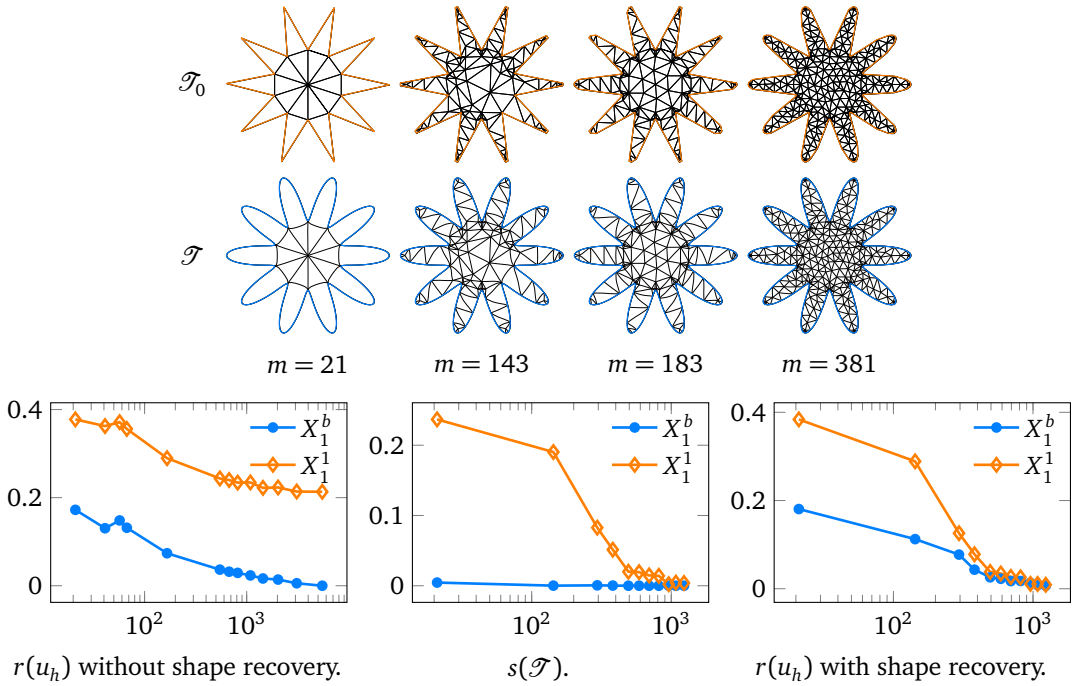
to estimate the approximation error [91].

In classical finite element simulations the original shape is usually not recovered when performing mesh refinement as shown in Figure 6.7. For this reason, $r(u_h)$ does not converge to zero for the standard solution, while our approach converges (left plots in Figures 6.8, 6.9, and 6.10).

In order to better understand this behaviour, we measure the actual geometric deviation with

$$s(\mathscr{T}) = \|1\|_{L^2(\mathscr{T})},$$

which corresponds to the volume of the mesh (note that $s(\mathscr{T})$ is computed by summing the entries of the mass-matrix). We compute the volume by means of numerical quadrature, which might introduce errors, since our discretization consists of warped elements. For the standard discretization, when refining the mesh without recovering the shape, the volume trivially stays constant. Hence, in order to have a fair comparison, we increase the shape accuracy while refining the mesh to ensure that the shape of the domain also converges to the exact



$r(u_h)$ without shape recovery. $s(\mathcal{T})$. $r(u_h)$ with shape recovery.

Figure 6.8. Source meshes $\mathcal{T}_0$ with boundary Θ_0 (first row), warped meshes $\mathcal{T}$ used by our method (second row), and convergence plots against different numbers of degrees of freedom m (last row).

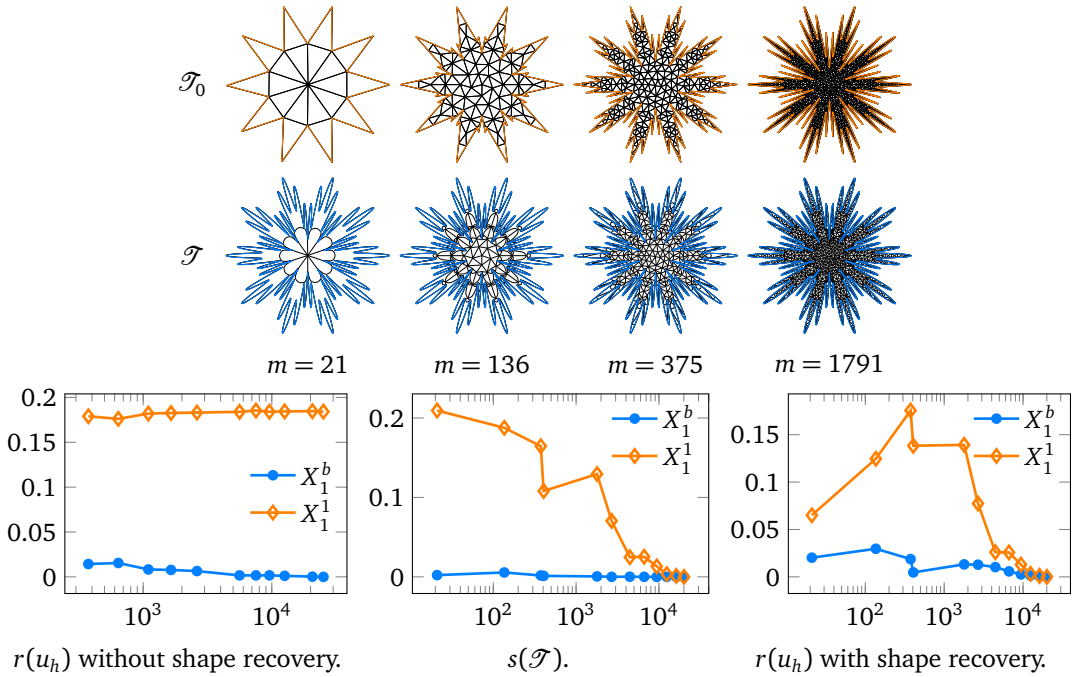


Figure 6.9. Source meshes $\mathscr{T}_0$ with boundary Θ_0 (first row), warped meshes $\mathscr{T}$ used by our method (second row), and convergence plots against different numbers of degrees of freedom m (last row).

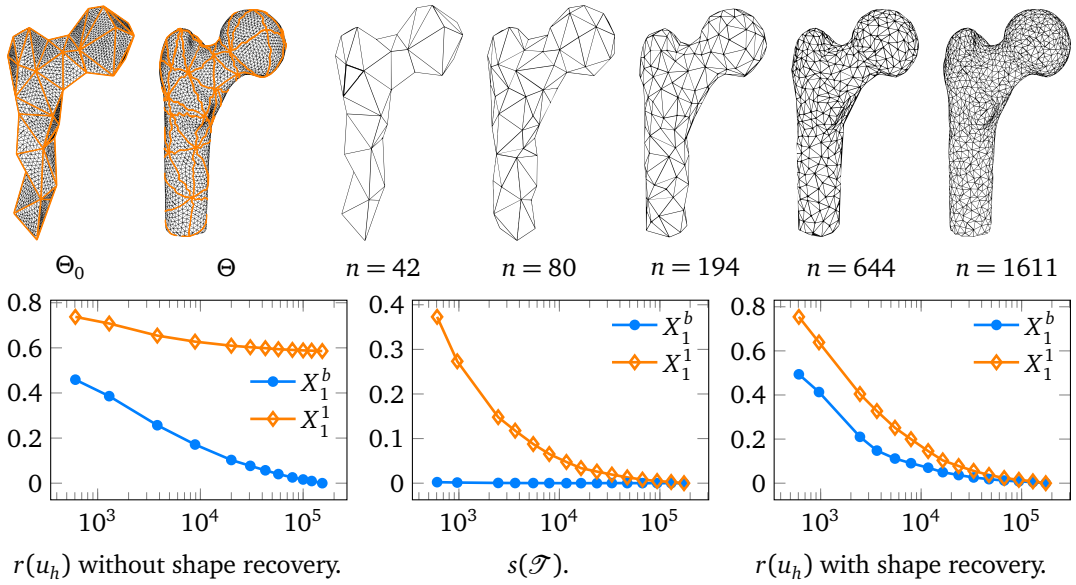


Figure 6.10. Convergence plots against different numbers of degrees of freedom m for a 3D experiment.

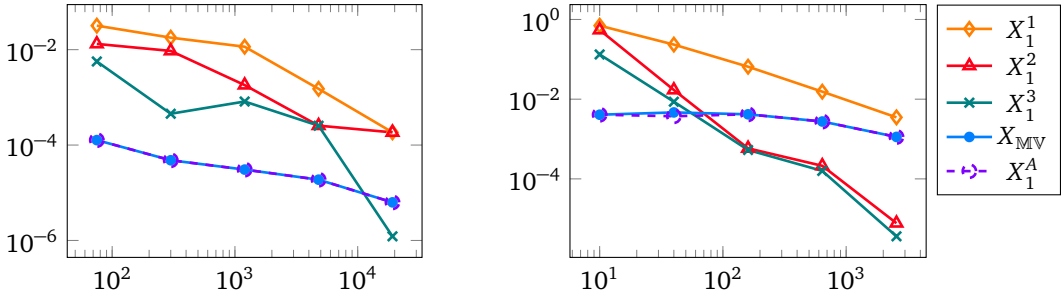


Figure 6.11. Estimation $s(\mathcal{T})$ of the volume error for different discretizations. The x-axis represents the number of elements, and the y-axis represents $s(\mathcal{T})$. The left plot describes the same experiment depicted in Figure 6.8, and right plot the experiment in Figure 6.9.

one. The behaviour of $s(\mathcal{T})$ shows that our discretization has almost zero geometrical error independently of h, while the standard discretization has higher geometrical error (middle plots in Figures 6.8, 6.9, and 6.10).

In order to investigate how the approximation error is influenced by the geometrical error, we measure $r(u_h)$ for our method and classical finite elements with shape recovery. Our discretization always has a smaller approximation error compared to the standard discretization (right plots in Figures 6.8, 6.9, and 6.10). This is due to the fact that our approach allows solving the problem in the exact geometry, even at low resolutions.

We performed the same experiments shown in Figure 6.8 and Figure 6.9 for the different piecewise approximations of b. In these experiments the standard discretization is represented by the iso-parameteric finite element discretizastion X_1^1, for which the geometric accuracy is increased together with the number of elements. In Figure 6.11 and Figure 6.12 we observe an improved convergence behaviour, in terms of geometric deviation $s(\mathcal{T})$ and estimation of the solution error $r(u_h)$, when employing polygonal elements and higher order piecewise polynomial map approximations. The local map approximations provide a computationally cheaper alternative to the discretizations built directly on b. The reason is that instead of evaluating b at each quadrature point in the assembly procedure we evaluate it only at each node of the mesh for constructing the approximation.

6.2.3 Conditioning

For solution methods such as iterative solvers, the condition number κ of the stiffness matrix plays an important role for the convergence rate [10]. In order

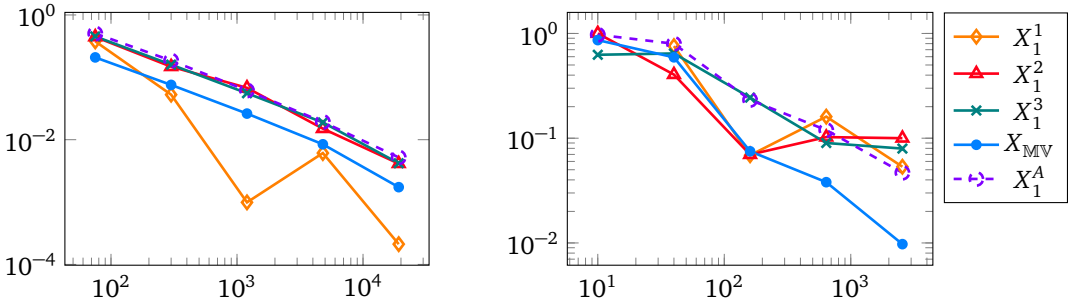


Figure 6.12. Estimation $r(u_h)$ of the solution error for different discretizations. The x-axis represents the number of elements, and the y-axis represents $r(u_h)$. The left plot describes the same experiment depicted in Figure 6.8, and right plot the experiment in Figure 6.9.

to understand how our discretization affects the condition number, we compute κ for the discrete Laplace operator $\boldsymbol{L}$ with respect to different mesh sizes h for both our discretization and the standard one. Because of the influence of the bijective mapping b, as shown in (4.6), our discretization has a slightly larger condition number. Figure 6.13 shows that $\kappa(\boldsymbol{L})$ behaves similarly for both discretizations which suggests that iterative solvers perform nearly as well for our discretization as for the standard one.

6.2.4 Convergence of the multigrid method with parametric finite elements

We observe the average convergence rate of the multigrid method applied to different parameterizations for reaching a residual $\mathbf{Mf} - \mathbf{Lu}$ with magnitude 10^{-12}. We compare it to a semi-geometric multigrid method where we construct the

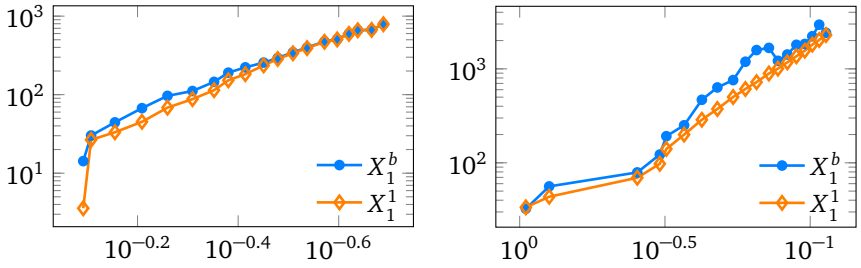


Figure 6.13. Condition number of the discrete Laplace operator $\kappa(\boldsymbol{L})$ against the mesh size h for the examples in Figure 6.8 (left) and Figure 6.9 (right).

coarse levels of the multigrid hierarchy by exploiting the pseudo-L^2-projection operator introduced in Section 2.2. For generating such hierarchy we first compute the axis-aligned bounding-box (AABB) of the input mesh $\mathscr{T} = \mathscr{T}^L$. Then, we compute $\mathscr{T}^1$ by meshing the AABB in such a way that the number of elements of $\mathscr{T}^1$ is smaller than the number of elements in $\mathscr{T}_L$ by a factor of $(2^d)^{L-1}$, where d is the spatial dimension. When building $\mathscr{T}^1$ we make sure that its elements have an aspect ratio close to one. We generate the intermediate $L-2$ meshes by (uniform) refinement of $\mathscr{T}^0$. Then, we compute the pseudo-L^2-projection operator $\mathbf{I}^L$ from the coarse space $V_h(\mathscr{T}_{L-1})$ to the fine space $V_h(\mathscr{T}_L)$ as explained in Section 2.3. We compute the prolongation operators for the lower levels as for standard geometric multigrid methods. We define this hierarchy of spaces as $\mathscr{H}_{\mathscr{P}}$. An overview of the geometric objects involved is shown in Figure 6.14.

Let us recall the spaces and hierarchies of the parametric discretization defined in Section 4.4. The hierarchy $\mathscr{H}_p^k$ is composed by the spaces $X_p^k(\mathscr{T}^l)$ where p represents the order of the polynomial basis functions defined in the reference element and k the polynomial order of the geometric transformation.

The hierarchy $\mathscr{H}_p^A$ is composed by the spaces X_p^A where p represents the order of the polynomial basis functions defined in the reference element and A represents the piece-wise geometric map defined for each element of the fine level mesh.

With $\mathscr{H}_{\mathbb{MV}}$ we denote the hierarchy of polygonal finite element spaces.

For our experiments we select different type of domains with several level of details, smooth and non-smooth features. However, as explained in Section 4.3.1 a valid piecewise k-th order polynomial map $\tilde{b}^k$ is not always available. Hence, we restrict our numerical evaluation to examples which allows constructing such map. We observe the average convergence rate

$$\rho = n^{-1} \sum_{q=1}^{n} \|\mathbf{L}\mathbf{u}^q - \mathbf{M}\mathbf{f}\| / \|\mathbf{L}\mathbf{u}^{q-1} - \mathbf{M}\mathbf{f}\|,$$

where $\mathbf{u}^q$ is the solution at the q-th iteration, n is the number of iterations, and $\mathbf{u}^0$ is the initial guess. Our observations are made with respect to different resolution of both the input mesh and the solution to the Poisson problem (4.1).

We observe that the hierarchies $\mathscr{H}_p^A$ and $\mathscr{H}_p^k$ consistently appear to provide the same convergence rate for all experiments (Figure 6.15 and Table 6.3). The hierarchies $\mathscr{H}_{\mathbb{MV}}$ and $\mathscr{H}_{\mathscr{P}}$ instead display degraded convergence rates depending on the geometric set-up. The loss of convergence appears to manifests itself for $\mathscr{H}_{\mathbb{MV}}$ when the shape of the elements is highly distorted for many layers around the boundary, and for $\mathscr{H}_{\mathscr{P}}$ when the domain has an extremely oscillatory

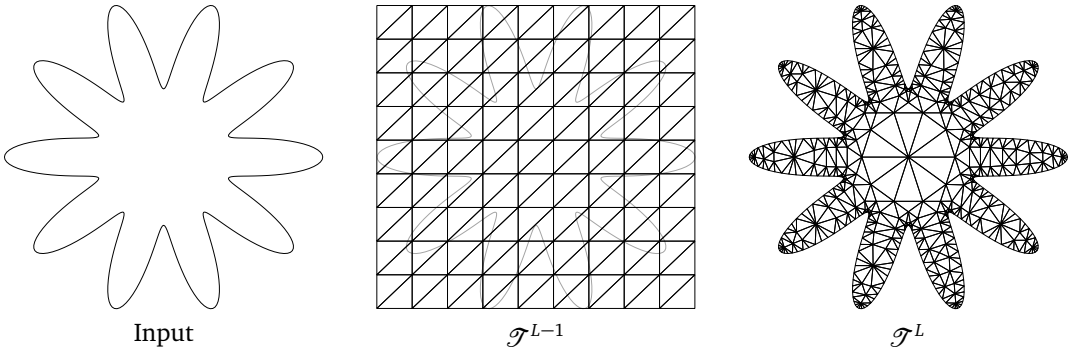


Figure 6.14. Example set-up of a geometric work-flow exploiting the L^2-projection for the construction of coarse spaces in multigrid hierarchies.

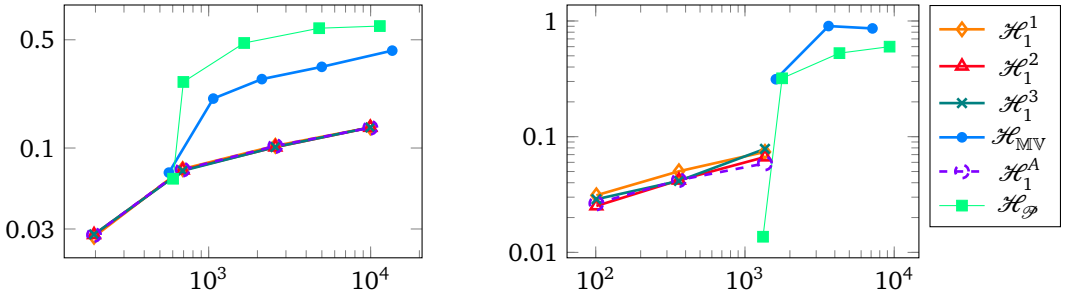


Figure 6.15. Average convergence rate ρ (y-axis) of the geometric multigrid method for different parameterizations, and mesh resolutions (x-axis). The left plot has the geometric set-up depicted in Figure 6.8, and right plot the set-up depicted in Figure 6.9.

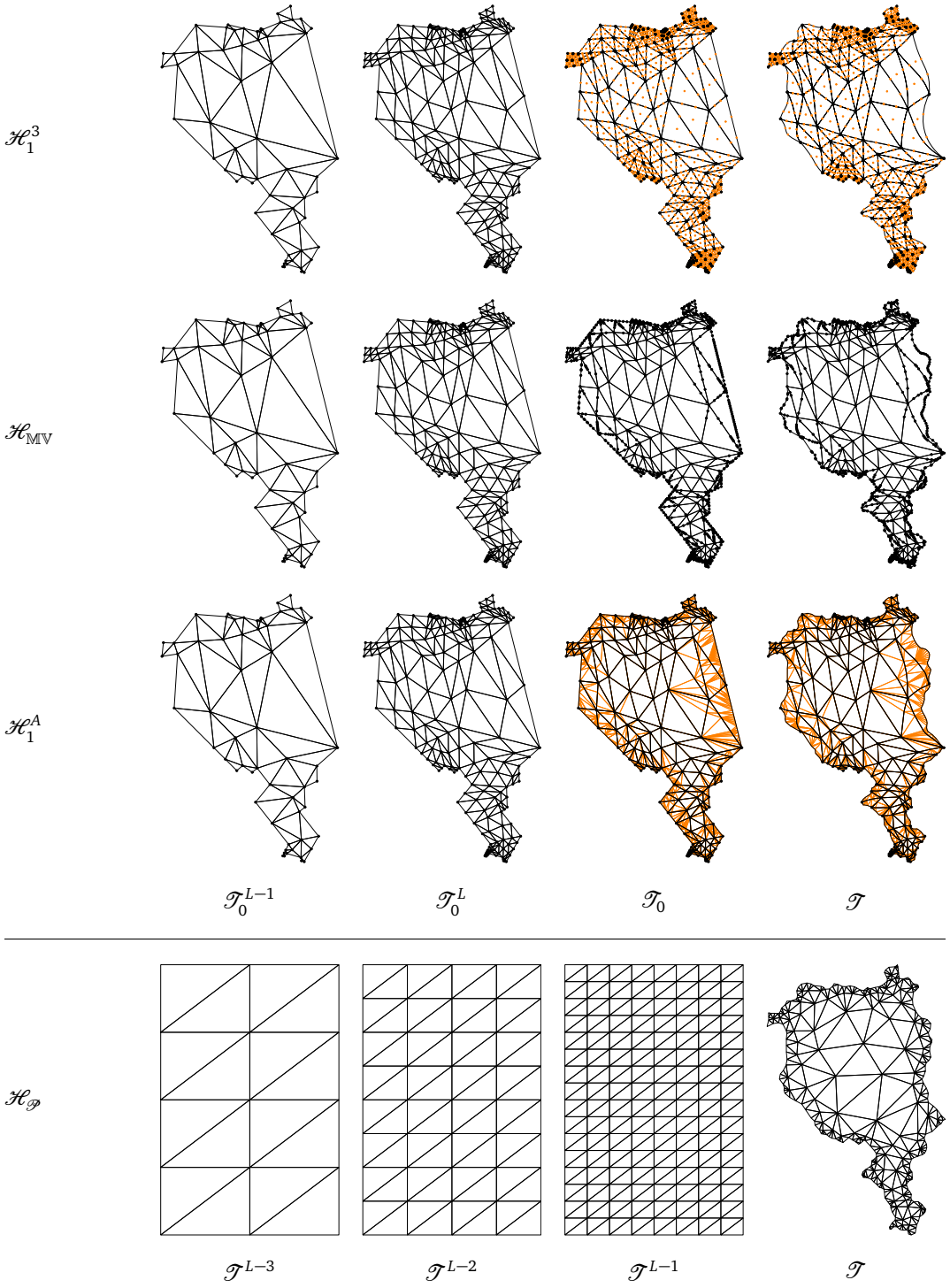


Figure 6.16. Finite element mesh hierarchies for different multilevel discretizations. The orange color markings highlight features of the geometric maps.

	# elements	# *d.o.f.* fine / coarse	# iterations	convergence rate ρ
$\mathscr{H}_1^1$	300	196 / 61	8	0.026
	1 200	691 / 196	11	0.073
	4 800	2 581 / 691	12	0.103
	19 200	9 961 / 2 581	14	0.135
$\mathscr{H}_1^2$ / $\mathscr{H}_1^3$	300	196 / 61	8	0.027
	1 200	691 / 196	11	0.071
	4 800	2 581 / 691	12	0.101
	19 200	9 961 / 2 581	14	0.135
$\mathscr{H}_{\mathbb{MV}}$	75	568 / 61	10	0.069
	300	1 068 / 196	18	0.208
	1 200	2 131 / 691	22	0.278
	4 800	4 993 / 2 581	25	0.334
	19 200	13 600 / 9 961	31	0.425
$\mathscr{H}_1^A$	300	196 / 61	8	0.027
	1 200	691 / 196	11	0.072
	4 800	2 581 / 691	12	0.103
	19 200	9 961 / 2 581	14	0.135
$\mathscr{H}_{\mathscr{P}}$	433	330 / 104	7	0.019
	669	448 / 153	31	0.406
	2 272	1 250 / 493	57	0.611
	6 999	3 637 / 1 881	57	0.613
	16 987	8 692 / 4 753	60	0.628

Table 6.3. Comparison of performance of the multigrid method with respect to different discretizations for the example shown in Figure 6.16.

shape (right plot Figure 6.15). Additionally, the convergence rate of the multigrid method is slightly worse when increasing the number of degrees of freedom for both $\mathscr{H}_{\mathbb{MV}}$ and $\mathscr{H}_{\mathscr{P}}$.

A more typical scenario and the different geometric approximations and mesh hierarchies are illustrated Figure 6.16. In this scenario, the multgrid method has convergence rates below 0.7 for all discretization, however the variants with parametric finite elements display a much better convergence behavior, as it can be observed in Table 6.3.

6.3 Chapter conclusion

We performed numerical experiments including weak-scaling and strong-scaling of our parallel algorithm for the variational transfer of discrete fields. We observed that most of the computational effort of our approach goes into computing the numerical quadrature, and for a large number of processes goes into finding intersection candidates. We investigated performance drivers. The time needed for communication of the actual geometric data is comparably small, and the main issue is the load-balancing which is challenged by the output-sensitive nature of the problem.

We studied the behaviour of our parametric finite element discretization based on mean-value mappings and its local approximations with respect to the Poisson problem. Through numerical experimentation we show that our super-parametric discretization generally has a lower approximation error compared to the standard one, due to the higher geometric accuracy, without significant changes on the conditioning of the discrete operators. We observed that our discretization does not affects the performance of the multigrid method for super-parametric case. For the case of polygonal finite elements based on mean-value coordinates we observed a shape dependent degrading behaviour.

Chapter 7

Conclusion

We investigated what we consider to be key issues related to complex geometric interactions in parallel multi-physics simulations. For dealing with such issues, we proposed a completely parallel strategy for transferring discrete fields between arbitrarily distributed finite element meshes and its applications. With a relatively small computational time overhead our strategy allows to simplify the simulation work-flow even for very complex mesh distribution scenarios. We studied the performance and the limitations of our strategy through detailed numerical experiments, and we provided several example application scenarios. We open-sourced and integrated our algorithms with the MFEM and the LIBMESH libraries.

We proposed a new parametric finite element discretization that allows decoupling the accuracy of the shape from the choice of the approximation space in finite element simulations. This separation allows for high flexibility with respect to the geometric objects in the simulation work-flow. We studied our discretization with several numerical experiments illustrating both promising results and limitations. Even if our discretization is based on mean-value mappings and their local approximations, we believe that further investigations may reveal more efficient and effective way of generating finite element discretizations.

We have discussed current trends and our idea for the development of scientific libraries. We instantiated our ideas with the UTOPIA library for which we provided a detailed description of its design and rationales. The UTOPIA library is public available as an open-source project.

A topic which has been only partially covered is the automatic determination of contact patches in contact problems. In fact, we have not covered this topic for parallel computations. In parallel settings the automatic determination of master and slave roles of contact patches may be a very useful tool which would

simplify the simulation work-flow significantly.

Bibliography

[1] M. AIGNER, C. HEINRICH, B. JÜTTLER, E. PILGERSTORFER, B. SIMEON, AND A.-V. VUONG, *Swept volume parameterization for isogeometric analysis*, Springer, 2009.

[2] P. R. AMESTOY, I. S. DUFF, J. KOSTER, AND J.-Y. L'EXCELLENT, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, 23 (2001), pp. 15–41.

[3] D. N. ARNOLD, F. BREZZI, B. COCKBURN, AND L. D. MARINI, *Unified analysis of discontinuous galerkin methods for elliptic problems*, SIAM Journal on Numerical Analysis, 39 (2002), pp. 1749–1779.

[4] P. AUBERT, N. DI CÉSARÉ, AND O. PIRONNEAU, *Automatic differentiation in c++ using expression templates and. application to a flow control problem*, Computing and Visualization in Science, 3 (2001), pp. 197–208.

[5] U. AYACHIT, *The ParaView guide : updated for ParaView version 4.3*, Kitware, 2015.

[6] M. BADER, *Space-filling curves: an introduction with applications in scientific computing*, vol. 9, Springer Science & Business Media, 2012.

[7] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202.

[8] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *Deal.ii & mdash; a general-purpose object-oriented finite element library*, ACM Transactions on Mathematical Software, 33 (2007).

[9] P. BASTIAN, G. BUSE, AND O. SANDER, *Infrastructure for the coupling of Dune grids*, in Proceedings of ENUMATH 2009, Springer, 2010, pp. 107–114.

[10] K.-J. BATHE AND E. L. WILSON, *Numerical methods in finite element analysis*, AMC, 10 (1976), p. 12.

[11] Y. BAZILEVS, C. MICHLER, V. CALO, AND T. HUGHES, *Isogeometric variational multiscale modeling of wall-bounded turbulent flows with weakly enforced boundary conditions on unstretched meshes*, Computer Methods in Applied Mechanics and Engineering, 199 (2010).

[12] H. BEN DHIA, *Multiscale mechanical problems: the Arlequin method*, Comptes Rendus de l'Academie des Sciences Series IIB Mechanics Physics Astronomy, 326 (1998), pp. 899–904.

[13] C. BERNARDI, Y. MADAY, AND F. RAPETTI, *Basics and some applications of the mortar element method*, GAMM-Mitt., 28 (2005), pp. 97–123.

[14] F. BERTRAND, S. MUÑZENMAIER, AND G. STARKE, *First-order system least squares on curved boundaries: Higher-order Raviart–Thomas elements*, SIAM Journal on Numerical Analysis, 52 (2014), pp. 3165–3180.

[15] F. BERTRAND, S. MUÑZENMAIER, AND G. STARKE, *First-order system least squares on curved boundaries: Lowest-order Raviart–Thomas elements*, SIAM Journal on Numerical Analysis, 52 (2014), pp. 880–894.

[16] J. BEY, *Tetrahedral grid refinement*, Computing, 55 (1995), pp. 355–378.

[17] D. BRAESS, *Finite elements. Theory, fast solvers and applications in solid mechanics*, Cambridge University Press, 2007.

[18] J. H. BRAMBLE, J. E. PASCIAK, AND O. STEINBACH, *On the stability of the L^2-projections in H^1*, Mathematics of Computation, 71 (2002), pp. 147–156.

[19] J. BRANDT, P. J. GUO, J. LEWENSTEIN, AND S. R. KLEMMER, *Opportunistic programming: How rapid ideation and prototyping occur in practice*, in Proceedings of the 4th International Workshop on End-user Software Engineering, ACM, 2008, pp. 1–5.

[20] S. BRENNER AND R. SCOTT, *The Mathematical Theory of Finite Element Methods*, vol. 15, Springer-Verlag New York, 2008.

[21] S. C. BRENNER AND I. TUTORIAL, *Geometric multigrid methods*, 2010.

[22] W. L. BRIGGS, V. E. HENSON, AND S. F. MCCORMICK, *A multigrid tutorial (2nd ed.)*, Society for Industrial and Applied Mathematics, 2000.

[23] Z. CAI AND G. STARKE, *Least-squares methods for linear elasticity*, SIAM Journal on Numerical Analysis, 42 (2004), pp. 826–842.

[24] P. CHAKRABORTY, Y. ZHANG, M. R. TONKS, AND S. B. BINER, *Multi-scale modeling of inter-granular fracture in uo2*, tech. rep., Idaho National Laboratory (INL), Idaho Falls, ID (United States), 2015.

[25] H. CHILDS, E. BRUGGER, B. WHITLOCK, J. MEREDITH, S. AHERN, D. PUGMIRE, K. BIAGAS, M. MILLER, C. HARRISON, G. H. WEBER, H. KRISHNAN, T. FOGAL, A. SANDERSON, C. GARTH, E. W. BETHEL, D. CAMP, O. RÜBEL, M. DURANT, J. M. FAVRE, AND P. NAVRÁTIL, *VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data*, in High Performance Visualization–Enabling Extreme-Scale Scientific Insight, Taylor and Francis, Oct 2012, pp. 357–372.

[26] P. G. CIARLET AND P.-A. RAVIART, *Interpolation theory over curved elements, with applications to finite element methods*, Computer Methods in Applied Mechanics and Engineering, 1 (1972), pp. 217–249.

[27] K. L. CLARKSON, R. E. TARJAN, AND C. J. VAN WYK, *A fast las vegas algorithm for triangulating a simple polygon*, Discrete & Computational Geometry, 4 (1989), pp. 423–432.

[28] M. COUR CHRISTENSEN, U. VILLA, A. ENGSIG-KARUP, AND P. VASSILEVSKI, *Nonlinear multigrid solver exploiting amge coarse spaces with approximation properties*, tech. rep., Lawrence Livermore National Laboratory, 2016.

[29] M. DE BERG, O. C. M. VAN KREVELD, AND M. OVERMARS, *Computational Geometry Algorithms and Applications*, Springer-Verlag Italia, 2008.

[30] A. DE BOER, A. VAN ZUIJLEN, AND H. BIJL, *Review of coupling methods for non-matching meshes*, Computer Methods in Applied Mechanics and Engineering, 196 (2007), pp. 1515–1525.

[31] Z. DEVITO, N. JOUBERT, F. PALACIOS, S. OAKLEY, M. MEDINA, M. BARRIENTOS, E. ELSEN, F. HAM, A. AIKEN, K. DURAISAMY, E. DARVE, J. ALONSO, AND P. HANRAHAN, *Liszt: A domain specific language for building portable mesh-based pde solvers*, in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, pp. 9:1–9:12.

[32] T. DICKOPF, *Multilevel methods based on non-nested meshes*, PhD thesis, Friedrich-Wilhelms University of Bonn, 2010.

[33] T. DICKOPF AND R. KRAUSE, *Efficient simulation of multi-body contact problems on complex geometries: A flexible decomposition approach using constrained minimization*, International Journal for Numerical Methods in Engineering, 77 (2009), pp. 1834–1862.

[34] T. DICKOPF AND R. KRAUSE, *Weak information transfer between non-matching warped interfaces*, in Domain Decomposition Methods in Science and Engineering XVIII, M. Bercovier, M. Gander, R. Kornhuber, and O. Widlund, eds., vol. 70, Springer, 2009, pp. 283–290.

[35] T. DICKOPF AND R. KRAUSE, *Monotone multigrid methods based on parametric finite elements*, tech. rep., ICS, USI, may 2011.

[36] T. DICKOPF AND R. KRAUSE, *Evaluating local approximations of the L^2-orthogonal projection between non-nested finite element spaces*, Numerical Mathematics: Theory, Methods, and Applications, 7 (2014).

[37] W. DÖRFLER AND M. RUMPF, *An adaptive strategy for elliptic problems including a posteriori controlled boundary approximation*, Mathematics of Computation of the American Mathematical Society, 67 (1998), pp. 1361–1382.

[38] G. DOS REIS, M. HALL, AND G. NISHANOV, *A module system for c++(revision 2)*, 2014.

[39] A. DÜRRBAUM, W. KLIER, AND H. HAHN, *Comparison of automatic and symbolic differentiation in mathematical modeling and computer simulation of rigid-body systems*, Multibody System Dynamics, 7 (2002), pp. 331–355.

[40] M. ECK, T. DEROSE, T. DUCHAMP, H. HOPPE, M. LOUNSBERY, AND W. STUETZLE, *Multiresolution analysis of arbitrary meshes*, in Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques, ACM, 1995, pp. 173–182.

[41] C. ERICSON, *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology)*, Morgan Kaufmann Publishers Inc., 2004.

[42] A. P. ERIKSON AND K. ÅSTRÖM, *Analysis for Science, Engineering and Beyond: The Tribute Workshop in Honour of Gunnar Sparr held in Lund, May 8-9, 2008*, Springer Berlin Heidelberg, 2012, pp. 93–141.

[43] A. ERN AND J.-L. GUERMOND, *Theory and practice of finite elements*, vol. 159, Springer Science & Business Media, 2013.

[44] L. C. EVANS, *Partial differential equations*, American Mathematical Society, 1998.

[45] P. E. FARRELL, *Galerkin projection of discrete fields via supermesh construction*, PhD thesis, Imperial College London, September 2009.

[46] P. E. FARRELL AND J. MADDISON, *Conservative interpolation between volume meshes by local galerkin projection*, Computer Methods in Applied Mechanics and Engineering, 200 (2011), pp. 89–100.

[47] B. FLEMISCH AND B. I. WOHLMUTH, *Stable Lagrange multipliers for quadrilateral meshes of curved interfaces in 3D*, Computer Methods in Applied Mechanics and Engineering, 196 (2007), pp. 1589–1602.

[48] M. S. FLOATER, *Mean value coordinates*, Computer Aided Geometric Design, 20 (2003), pp. 19–27.

[49] M. J. GANDER AND C. JAPHET, *An algorithm for non-matching grid projections with linear complexity*, in Domain Decomposition Methods in Science and Engineering XVIII, Springer, 2009, pp. 185–192.

[50] D. GASTON, C. NEWMAN, G. HANSEN, AND D. LEBRUN-GRANDIE, *Moose: A parallel computational framework for coupled systems of nonlinear equations*, Nuclear Engineering and Design, 239 (2009), pp. 1768–1778.

[51] F. H. GEISLER, *The CHARITE artificial disc: design history, FDA IDE study results, and surgical technique.*, Clinical neurosurgery, 53 (2006), pp. 223–228.

[52] F. GRECO AND N. SUKUMAR, *Derivatives of maximum-entropy basis functions on the boundary: Theory and computations*, International Journal for Numerical Methods in Engineering, 94 (2013), pp. 1123–1149.

[53] D. GROEN, S. J. ZASADA, AND P. V. COVENEY, *Survey of multiscale and multiphysics applications and communities*, Computing in Science & Engineering, 16 (2014), pp. 34–43.

[54] C. GROSS AND R. KRAUSE, *On the convergence of recursive Trust–Region methods for multiscale non-linear optimization and applications to nonlinear mechanics*, SIAM Journal on Numerical Analysis, 47 (2009), pp. 3044–3069.

[55] G. GUENNEBAUD, B. JACOB, ET AL., *Eigen v3*, 2010.

[56] B. GUENTER, *Efficient symbolic differentiation for graphics applications*, ACM Transactions on Graphics, 26 (2007), p. 108.

[57] W. HACKBUSCH, *Multi-grid methods and applications*, vol. 4, Springer-Verlag, 1985.

[58] M. HEROUX, R. BARTLETT, V. H. R. HOEKSTRA, J. HU, T. KOLDA, R. LEHOUCQ, K. LONG, R. PAWLOWSKI, E. PHIPPS, A. SALINGER, H. THORNQUIST, R. TUMINARO, J. WILLENBRING, AND A. WILLIAMS, *An overview of trilinos*, Tech. Rep. SAND2003–2927, Sandia National Laboratories, 2003.

[59] M. A. HEROUX, J. M. WILLENBRING, AND R. HEAPHY, *Trilinos developers guide*, Tech. Rep. SAND2003–1898, Sandia National Laboratories, 2007.

[60] C. HESCH AND P. BETSCH, *A comparison of computational methods for large deformation contact problems of flexible bodies*, ZAMM - Journal of Applied Mathematics and Mechanics, 86 (2006), pp. 818–827.

[61] C. HESCH AND P. BETSCH, *Transient three-dimensional domain decomposition problems: Frame-indifferent mortar constraints and conserving integration*, International Journal for Numerical Methods in Engineering, 82 (2010), pp. 329–358.

[62] C. HESCH AND P. BETSCH, *Isogeometric analysis and domain decomposition methods*, Computer Methods in Applied Mechanics and Engineering, 213 (2012), pp. 104–112.

[63] C. HESCH AND P. BETSCH, *An object oriented framework: From flexible multibody dynamics to fluid-structure interaction*, The 2nd Joint International Conference on Multibody System Dynamics, (2012).

[64] C. HESCH, A. GIL, A. A. CARREÑO, J. BONET, AND P. BETSCH, *A mortar approach for fluid–structure interaction problems: Immersed strategies for deformable and rigid bodies*, Computer Methods in Applied Mechanics and Engineering, 278 (2014), pp. 853–882.

[65] M. HOLST, *FEtk: Finite Element ToolKit*, 2003.

[66] K. HORMANN AND N. SUKUMAR, *Maximum entropy coordinates for arbitrary polytopes*, Computer Graphics Forum, 27 (2008), pp. 1513–1520.

[67] T. J. HUGHES, J. A. COTTRELL, AND Y. BAZILEVS, *Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement*, Computer Methods in Applied Mechanics and Engineering, 194 (2005), pp. 4135–4195.

[68] J. R. HUMPHREY, D. K. PRICE, K. E. SPAGNOLI, A. L. PAOLINI, AND E. J. KELMELIS, *Cula: hybrid gpu accelerated linear algebra routines*, in SPIE defense, security, and sensing, 2010, pp. 770502–770502.

[69] K. IGLBERGER, G. HAGER, J. TREIBIG, AND U. RÜDE, *Expression templates revisited: a performance analysis of current methodologies*, SIAM Journal on Scientific Computing, 34 (2012), pp. C42–C69.

[70] A. JACOBSON, *Bijective mappings with generalized barycentric coordinates: a counterexample*, tech. rep., Department of Computer Science, ETH Zurich, 2012.

[71] P. JOLIVET, V. DOLEAN, F. E. E. HECHT, F. E. E. NATAF, C. PRUD HOMME, AND N. SPILLANE, *High performance domain decomposition methods on massively parallel architectures with freefem++*, Journal of Numerical Mathematics, 20 (2012), pp. 287–302.

[72] W. JOPPICH AND M. KÜRSCHNER, *MpCCI - a tool for the simulation of coupled applications*, Concurrency and Computation: Practice and Experience, 18 (2006), pp. 183–192.

[73] T. JU, S. SCHAEFER, AND J. WARREN, *Mean value coordinates for closed triangular meshes*, ACM Transactions on Graphics, 24 (2005), pp. 561–566.

[74] KHRONOS OPENCL WORKING GROUP, *The OpenCL Specification, version 1.0.29*, 2008.

[75] N. KIKUCHI AND J. T. ODEN, *Contact problems in elasticity: a study of variational inequalities and finite element methods*, vol. 8, siam, 1988.

[76] 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 (2006), pp. 237–254.

[77] T. KOLEV, *MFEM: Modular finite element methods*, 2016.

[78] D. KRAUSE, K. FACKELDEY, AND R. KRAUSE, *A parallel multiscale simulation toolbox for coupling molecular dynamics and finite elements*, in Singular Phenomena and Scaling in Mathematical Models, M. Griebel, ed., Springer International Publishing, 2014, pp. 327–346.

[79] D. KRAUSE AND R. KRAUSE, *MACI*. A Parallel Multiscale Simulation Toolbox for Coupling Molecular Dynamics and Finite Elements, 2014.

[80] R. KRAUSE AND O. SANDER, *Automatic construction of boundary parametrizations for geometric multigrid solvers*, Computing and Visualization in Science, 9 (2006), pp. 11–22.

[81] R. KRAUSE AND P. ZULIAN, *A parallel approach to the variational transfer of discrete fields between arbitrarily distributed finite element meshes*, SIAM Journal on Scientific Computing, 38 (2016), pp. C307–C333.

[82] B. P. LAMICHHANE, R. P. STEVENSON, AND B. I. WOHLMUTH, *Higher order mortar finite element methods in 3D with dual Lagrange multiplier bases*, Numerische Mathematik, 102 (2005), pp. 93–121.

[83] A. W. LEE, W. SWELDENS, P. SCHRÖDER, L. COWSAR, AND D. DOBKIN, *Maps: Multiresolution adaptive parameterization of surfaces*, in Proceedings of the 25th annual conference on Computer graphics and interactive techniques, 1998, pp. 95–104.

[84] S. LEFEBVRE AND H. HOPPE, *Perfect spatial hashing*, in ACM SIGGRAPH 2006 Papers, ACM, 2006, pp. 579–588.

[85] B. LI, X. LI, K. WANG, AND H. QIN, *Surface mesh to volumetric spline conversion with generalized polycubes*, IEEE Transactions on Visualization and Computer Graphics, 19 (2013), pp. 1539–1551.

[86] Q. LI, K. ITO, Z. WU, C. S. LOWRY, I. LOHEIDE, AND P. STEVEN, *Comsol multiphysics: A novel approach to ground water modeling*, Groundwater, 47 (2009), pp. 480–487.

[87] H. LIAN, S. P. A. BORDAS, R. SEVILLA, AND R. N. SIMPSON, *Recent Developments in CAD/analysis Integration*, ArXiv e-prints, (2012).

[88] A. LOGG, *Automating the finite element method*, Archives of Computational Methods in Engineering, 14 (2007), pp. 93–138.

[89] A. LOGG AND G. N. WELLS, *DOLFIN: automated finite element computing*, CoRR, abs/1103.6248 (2011).

[90] K. LONG, R. KIRBY, AND B. VAN BLOEMEN WAANDERS, *Unified embedded parallel finite element computations via software-based frÃ©chet differentiation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3323–3351.

[91] X. LUO, M. S. SHEPHARD, AND J.-F. REMACLE, *The influence of geometric approximation on the accuracy of high order methods*, Rensselaer SCOREC report, 1 (2001).

[92] R. C. MARTIN, *More C++ gems*, vol. 17, Cambridge University Press, 2000.

[93] T. MARTIN AND E. COHEN, *Volumetric parameterization of complex objects by respecting multiple materials*, Computers & Graphics, 34 (2010), pp. 187–197.

[94] A. MASSING, M. G. LARSON, A. LOGG, AND M. E. ROGNES, *An overlapping mesh finite element method for a fluid-structure interaction problem*, arXiv preprint, (2013).

[95] J. MELENK AND B. WOHLMUTH, *On residual-based a posteriori error estimation in hp-fem*, Advances in Computational Mathematics, 15 (2001), pp. 311–331.

[96] B. MEYER, *Applying design by contract*, IEEE Transactions on Computers, 25 (1992), pp. 40–51.

[97] C. MIEHE, F. WELSCHINGER, AND M. HOFACKER, *Thermodynamically consistent phase-field models of fracture: Variational principles and multi-field fe implementations*, International Journal for Numerical Methods in Engineering, 83 (2010).

[98] J. MOSLER AND M. ORTIZ, *Variational h-adaption in finite deformation elasticity and plasticity*, International Journal for Numerical Methods in Engineering, 72 (2007), pp. 505–523.

[99] J. NICKOLLS, I. BUCK, M. GARLAND, AND K. SKADRON, *Scalable parallel programming with CUDA*, Queue, 6 (2008), pp. 40–53.

[100] C. NVIDIA, *Cublas library*, 2008.

[101] J. O'ROURKE, *Computational Geometry in C*, Cambridge University Press, 1998.

[102] A. PEHLIVANOV, G. CAREY, AND P. VASSILEVSKI, *Least-squares mixed finite element methods for non-selfadjoint elliptic problems: I. error estimates*, Numerische Mathematik, 72 (1996), pp. 501–522.

[103] K. B. PETERSEN, M. S. PEDERSEN, ET AL., *The matrix cookbook*, 2008.

[104] C. PRUD HOMME, V. CHABANNES, V. DOYEUX, M. ISMAIL, A. SAMAKE, AND G. PENA, *Feel++: A computational framework for galerkin methods and advanced numerical methods*, in ESAIM: Proceedings, vol. 38, 2012, pp. 429–455.

[105] M. A. PUSO, *A 3D mortar method for solid mechanics*, International Journal for Numerical Methods in Engineering, 59 (2004).

[106] M. A. PUSO AND T. A. LAURSEN, *A mortar segment-to-segment contact method for large deformation solid mechanics*, Computer Methods in Applied Mechanics and Engineering, 193 (2004).

[107] L. QI AND J. SUN, *A nonsmooth version of newton's method*, Mathematical programming, 58 (1993), pp. 353–367.

[108] A. QUARTERONI, *Numerical Models for Differential Problems*, Springer, 2009.

[109] A. QUARTERONI AND A. VALLI, *Domain decomposition methods for partial differential equations*, Clarendon Press, 1999.

[110] M. RANDRIANARIVONY, *Tetrahedral transfinite interpolation with b-patch faces: construction and regularity*, INS Preprint, 803 (2008).

[111] M. RANDRIANARIVONY, *On transfinite interpolations with respect to convex domains*, Computer Aided Geometric Design, 28 (2011), pp. 135–149.

[112] F. RATHGEBER, D. A. HAM, L. MITCHELL, M. LANGE, F. LUPORINI, A. T. T. MCRAE, G. BERCEA, G. R. MARKALL, AND P. H. J. KELLY, *Firedrake: automating the finite element method by composing abstractions*, CoRR, abs/1501.01809 (2015).

[113] S. P. REISS, *The challenge of helping the programmer during debugging*, in Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on, 2014, pp. 112–116.

[114] S. L. RIDGWAY, *Finite element techniques for curved boundaries*, PhD thesis, Massachusetts Institute of Technology, 1973.

[115] S. L. RIDGWAY, *Interpolated boundary conditions in the finite element method*, SIAM Journal on Numerical Analysis, 12 (1975), pp. 404–427.

[116] M. RIVI, L. CALORI, G. MUSCIANISI, AND V. SLAVNIC, *In-situ visualization: State-of-the-art and some use cases*, PRACE White Paper, (2012).

[117] K. RUPP, *Gpu-accelerated non-negative matrix factorization for text mining*, in NVIDIA GPU Technology Conference, 2012, p. 77.

[118] K. RUPP, F. RUDOLF, AND J. WEINBUB, *A discussion of selected vienna-libraries for computational science*, 2013.

[119] C. SANDERSON AND R. CURTIN, *Armadillo: a template-based C++ library for linear algebra*, The Journal of Open Source Software, 1 (2016).

[120] T. SCHNEIDER AND K. HORMANN, *Smooth bijective maps between arbitrary planar polygons*, Computer Aided Geometric Design, (2015).

[121] T. SCHNEIDER, K. HORMANN, AND M. S. FLOATER, *Bijective composite mean value mappings*, Computer Graphics Forum, 32 (2013), pp. 137–146.

[122] T. SCHNEIDER, P. ZULIAN, M. R. AZADMANESH, R. KRAUSE, AND M. HAUSWIRTH, *Vestige: A visualization framework for engineering geometry-related software*, Proceedings of VISSOFT (3rd IEEE Working Conference on Software Visualization), (2015).

[123] R. SEIDEL, *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, Computational Geometry, 1 (1991), pp. 51–64.

[124] R. SEVILLA, S. FERNÁNDEZ-MÉNDEZ, AND A. HUERTA, *Nurbs-enhanced finite element method (nefem)*, Archives of Computational Methods in Engineering, 18 (2011), pp. 441–484.

[125] J. SHEWCHUK, *What is a good linear finite element? interpolation, conditioning, anisotropy, and quality measures (preprint)*, University of California at Berkeley, 73 (2002).

[126] J. R. SHEWCHUK, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[127] A. H. STROUD AND D. SECREST, *Gaussian Quadrature Formulaes*, Prentice-Hall, 1966.

[128] N. SUKUMAR AND E. A. MALSCH, *Recent advances in the construction of polygonal finite element interpolants*, Archives of Computational Methods in Engineering, 13 (2006).

[129] H. SUNDAR, R. S. SAMPATH, AND G. BIROS, *Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708.

[130] I. E. SUTHERLAND AND G. W. HODGMAN, *Reentrant polygon clipping*, Commun. ACM, 17 (1974), pp. 32–42.

[131] J. TSUDA, *Practical rigid body physics for games*, in ACM SIGGRAPH ASIA 2009 Courses, ACM, 2009, pp. 14:1–14:83.

[132] L. R. TURNER, *Inverse of the vandermonde matrix with applications*, NASA technical note D-3547, (1996).

[133] T. VELDHUIZEN, *Expression templates*, 1995.

[134] J. WALTER AND M. KOCH, *BOOST C++ Libraries*, 2009.

[135] Q. WANG, H. ZHOU, AND D. WAN, *Numerical simulation of wind turbine blade-tower interaction*, Journal of Marine Science and Application, 11 (2012), pp. 321–327.

[136] J. WLOKA, *Partial differential equations*, Cambridge university press, 1987.

[137] B. I. WOHLMUTH, *A mortar finite element method using dual spaces for the Lagrange multiplier*, SIAM Journal on Numerical Analysis, 38 (1998), pp. 989–1012.

[138] B. I. WOHLMUTH AND R. H. KRAUSE, *Monotone multigrid methods on non-matching grids for nonlinear multibody contact problems*, SIAM Journal on Scientific Computing, 25 (2003), pp. 324–347.

[139] S. XIAO AND T. BELYTSCHKO, *A bridging domain method for coupling continua with molecular dynamics*, Computer Methods in Applied Mechanics and Engineering, 193 (2004), pp. 1645–1669.

[140] D. XUE, L. DEMKOWICZ, ET AL., *Control of geometry induced error in hp finite element (fe) simulations. i. evaluation of fe error for curvilinear geometries*, Int. J. Numer. Anal. Model, 2 (2005), pp. 283–300.

[141] B. YANG AND T. A. LAURSEN, *A large deformation mortar formulation of self contact with finite sliding*, Computer Methods in Applied Mechanics and Engineering, 197 (2008), pp. 756–772.

[142] P. ZULIAN, A. KOPANIČÁKOVÁ, AND T. SCHNEIDER, *Utopia: A c++ embedded domain specific language for scientific computing*, 2016.