Geometry decomposition algorithms for the Nitsche method on unfitted geometries

by

TOM ANDREAS NÆRLAND

THESIS
for the degree of
MASTER OF SCIENCE

(Master i Anvendt matematikk og mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2014

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

I Acknowledgement

I want to thank my supervisor André Massing at Simula Research Laboratory for his patience and large amount of insight he has lent me in his field. A big thanks also goes to my fellow master students Jarle, Ingeborg and Nina for the company and good discussions. Thanks also to my friends Bjørnar, Live, Hallvar and Kaja and my brother Martin and mom for the steadfast support they have been these last semesters.

Contents

Chapter 1

Introduction

Recent developments in the fields of medical imaging and computational science makes it increasingly more feasible to do numerical simulation on geometries derived from medical scans. The view this opens into medical data can be a valuable tool in reaching new understandings of the physical processes governing life and health. As these methods hopefully mature, numerical simulation on patient-specific geometries has the potential to usher in a new age of individualized clinical treatments based on medical diagnostic methodology both more informative and accurate than that of the present day.

The development of multidisciplinary applications such as mentioned above naturally requires a combination of computational frameworks, each of which are mature and robust. A common framework for numerical simulation of physical systems, in this regard, is *finite element analysis* of partial differential equations. Finite element analysis has proven to be a framework of great utility in fields such as engineering, medicine and economics and is actively developed towards an increasing number of both current and future applications.

One of the major problems in finite element analysis is automating the process of turning a complex mathematical formulation into a problem that can be posed and solved on a computer. In recent years, much research has been carried out on automating both the generation of efficient numerical schemes for many problems and efficiently solving the resulting numerics. The computational models coupling with the geometrical description of its domain, however, has arguably not been given the same extensive treatment. Although finite element analysis originates in structural engineering, it is estimated that for complex engineering designs, around 80% [1] of the total time in the design and simulation process is spent on translating geometrical descriptions such as CAD (Computer Aided Design) models into computational domains that can be passed on to finite element analysis codes. For biomedical applications, this process, known as *meshing* a domain, is equivalently difficult and time consuming; here with the added challenge of having to represent tissues of far greater complexity than that of steel and polymer as associated with structural engineering.

At the present moment, many flexible geometrical approaches to finite element analysis are being explored. Notable lately is the development of *isogeometric analysis* [2] which enables finite element analysis directly on CAD geometry. Parallel to this, research on other methods for flexible geometrical descriptions is also gaining ground, particularly with the development of finite element methods for what is here referred to as *unfitted* geometries. These methods allow much of the existing catalogue of well known finite element theory to be applied on domains where *some* of the traditional geometrical restrictions are removed. These approaches are based on coupling methods like *Lagrange multiplier methods* [3, 4], *discontinuous Galerkin methods* [5, 6, 7, 8] and *immersed boundary methods* [9, 10], to name just a few. The finite element methodology presented in this thesis uses *the Nitsche method* [11] to weakly impose boundary and interface conditions to in turn ease some typical geometrical restrictions, with relatively few extensions to the standard FEM. The finite element analysis with the Nitsche method is here in large parts based on work by Massing et al. [12, 13, 14], Hansbo and Burman [15, 16, 3] and Stenberg and Jutunen [17]. This method, along with specific applications wherein it can be of good use, is further introduced in Chapter 3.

Apart from presenting some of the mentioned finite element theory, the primary concern of this thesis is deriving geometrical entities suitable for finite element analysis. Many algorithms from the field of computational geometry are reviewed and implemented to enable the solution of partial differential equations on unfitted geometries. A core example of an unfitted geometry, forming the practical foundation of much of this thesis, is that of two *meshes* arbitrarily allowed to overlap and together constitute a full computational domain. This setting is illustrated in Figure 1.1b. The codes described and developed in this thesis are primarily intended for this *overlapping meshes* case, but is in overall concept and implementation attempted to be both general and extensible to many other types of computational domains. To this end, departures are made into conceptual descriptions and implementations of several algorithms fit for this application and used prior in similar settings.

As it happens, many of the researchers in this field use entirely different techniques from the field of computational geometry to achieve many of the same geometrical goals. Techniques ranging from simple *check-all-against-all* brute force methods to external libraries designed for either exceptional numerical robustness [18] or exceptional computational speed [19, 20, 21], are all used to derive geometrical information needed by different finite element methods. Many of these approaches are here presented, along with implementations and investigations into the specifics of some of them. Particularly this thesis launches an investigation into which *bounding volume* is best to use in a *bounding volume hierarchy* [20, 22] for detecting intersections in meshes typical for finite element analysis. This hierarchy is further compared against another efficient intersection algorithm, an *advancing front algorithm* [21]. At the end of Chapter 4: *Collision detection and mesh decomposition*, an algorithm based on the widely used *marching cubes* algorithm [23] is presented and implemented for mesh decomposition.

This thesis is organized as follows. In Chapter 2: *The finite element method on fitted meshes*, preliminaries are presented for a standard finite element method. Here a finite element method is derived for the Poisson problem on a standard mesh geometry. In Chapter 3: *The finite element method on unfitted geometries*, selected applications benefiting from a flexible geometrical approach are presented and the Nitsche method is formulated for the application at hand. Chapter 4: *Collision detection and mesh decomposition* is devoted to deriving the geometric intersection information and geometrical entities needed for the Nitsche method in the overlapping meshes case, which is a major subject of study in this thesis. Chapter 5: *Integration and assembly of cut elements* presents ways to integrate on cut geometries and presents the assembly routine performed by the library LibCutFEM [14] as it is used here. Chapter 6: *Numerical results* presents a solution of the Poisson problem on overlapping meshes and a selection of associated convergence tests and timings. Chapter 7: *Conclusions and further work* summarizes challenges in implementation, results and further work.

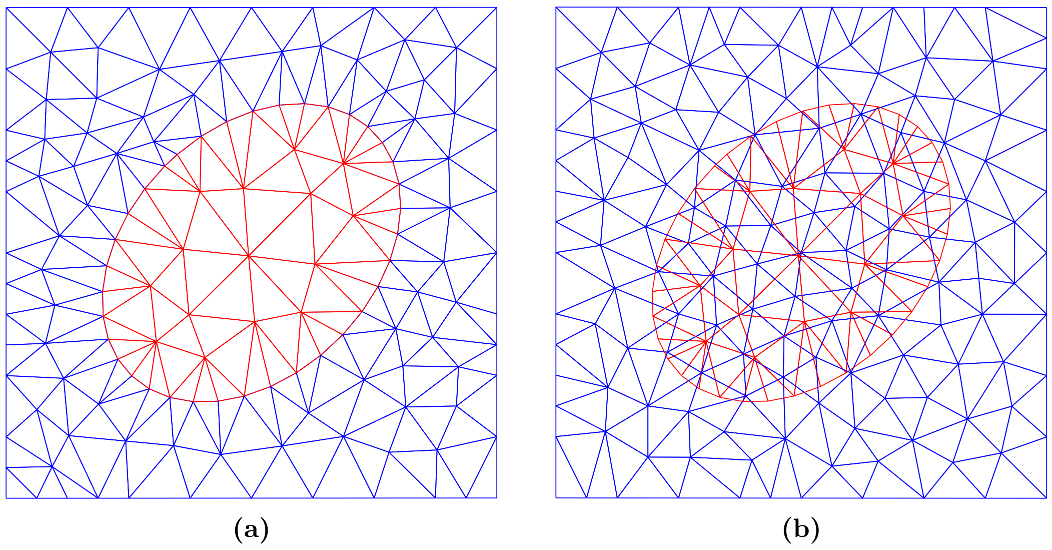


Figure 1.1: Examples of what is here denoted *fitted* (a) and *unfitted* geometry. Efficiently decomposing the geometry of *overlapping meshes* (b) into geometrical entities suitable for finite element analysis is a main objective of this thesis.

Chapter 2

The finite element method on fitted meshes

Natural phenomena such as waves, heat, fluid flow, elasticity, electrodynamics or quantum mechanics can all be described by partial differential equations. Formalized in similar terms in a PDE, many aspects of these physical systems can be studied with great efficiency. The equations constituting these descriptions are however often difficult to solve. Analytical solutions to PDEs are often hard to find, and in many cases, exact solutions do not exist. Approximating solutions to PDEs by numerical methods is thus often the only practical approach; one such method is derived in the following.

2.1 Model problem

The *Poisson's equation* is a simple model of many physical processes in its own right and an essential building block for a large number of more complex models. Here it is an apt point of departure for studying numerical approximation of partial differential equations. Let Ω be a bounded domain in $\mathbb{R}^d$, and $\partial\Omega$ denote its boundary. The Poisson's equation reads: find u such that

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= g_D && \text{on } \partial\Omega_D, \\ \nabla u \cdot \mathbf{n} &= g_N && \text{on } \partial\Omega_N \end{aligned} \tag{2.1}$$

where $\mathbf{n}$ denotes the outward pointing normal of Ω. and $\partial\Omega_D \subseteq \partial\Omega$ and $\partial\Omega_N \subseteq \partial\Omega$ denotes the *Dirichlet* and *Neumann* boundaries, respectively. The Dirichlet boundary condition, $u = g_D$, assigns a prescribed value of the unknown function u at $\partial\Omega_D$, while the Neumann boundary condition $\nabla u \cdot \mathbf{n} = g_N$ assigns a value for the normal derivative of u on $\partial\Omega_N$. The Laplacian operator Δ is the sum of all unmixed second partial derivatives of u, $\Delta u = \sum_{i=1}^{n} \partial^2 u/\partial x_i^2$.

In what follows a *finite element method* is developed for this problem, initiated by deriving a *continuous variational formulation* for (2.1).

2.1.1 Continuous variational formulation

A variational formulation for the model problem is derived by requiring that for a suitable *test space* V_g and *test function* $v \in V_g$,

$$\int_\Omega (\Delta u + f)v \, \mathrm{d}\mathbf{x} = 0. \tag{2.2}$$

This equation holds provided that the integrals are well defined, and it is readily seen that a solution u of (2.1) also satisfies (2.2). Now, by assuming v is sufficiently smooth, the smoothness required for u can be reduced by the property of the derivative in integration by parts and the divergence theorem, yielding

$$\int_\Omega \nabla u \nabla v \, \mathrm{d}\mathbf{x} = \int_\Omega f v \, \mathrm{d}\mathbf{x} + \int_{\partial\Omega} (\nabla u \cdot \mathbf{n}) v \, \mathrm{dS}.$$

where dS is used to distinguish the surface integral over $\partial\Omega$ from the volume integral over Ω. By replacing the Neumann-boundary term with its prescribed value, this is

$$\int_\Omega \nabla u \nabla v \, \mathrm{d}\mathbf{x} = \int_\Omega f v \, \mathrm{d}\mathbf{x} + \int_{\partial\Omega} g_N v \, \mathrm{dS}, \tag{2.3}$$

which is a continuous variational formulation of (2.1).

If u solves (2.3), it is a *weak solution* of the original problem. This solution is not necessarily smooth enough to solve (2.1) itself, but it can still retain alot of valuable information about the original problem. So, in what sense can the the weak solution and the test function v be meaningful? This is mainly a question of *where* to look for the solution u. Looking at the space of all functions that are square-integrable

$$L_2(\Omega) = \left\{ u : \Omega \to \mathbb{R} \;\middle|\; \int_\Omega u^2 \, \mathrm{d}\mathbf{x} < \infty \right\},$$

with the norm

$$||u||_{L_2} = \left(\int_\Omega u^2 \, \mathrm{d}\mathbf{x} \right)^{\frac{1}{2}},$$

it is clear that the integral on the left-hand side (2.3) will be well defined if all first derivatives are in $L_2(\Omega)$. This follows from using the Cauchy-Schwarz inequality and finding that

$$\int_\Omega \nabla u \nabla v \, \mathrm{d}\mathbf{x} = \sum_{i=1}^{d} \int_\Omega \left(\frac{\partial u}{\partial x_i} \right) \left(\frac{\partial v}{\partial x_i} \right) \mathrm{d}\mathbf{x} \;\leq\; \sum_{i=1}^{d} \left|\left| \frac{\partial u}{\partial x_i} \right|\right|_{L_2} \left|\left| \frac{\partial u}{\partial x_i} \right|\right|_{L_2} \leq \infty.$$

The integrals on the right-hand side of (2.3) is similarly well defined if $f, g \in L_2(\Omega)$. The space wherein the solution to (2.6) and the test function v naturally exists, is called the *Sobolev space* $H^1(\Omega)$, which for $\Omega \subset \mathbb{R}^d$ is defined as

$$H^1(\Omega) := \left\{ u : \Omega \to \mathbb{R} \;\middle|\; u, \frac{\partial u}{\partial x_i} \text{ for } i = 1, 2..., d \in L^2(\Omega) \right\}.$$

where the H^1 *semi-norm* is defined as

$$|u|_{H^1(\Omega)} = \left(\int_\Omega |\nabla(u)|^2 \, \mathrm{d}\mathbf{x} \right)^{\frac{1}{2}},$$

and the H^1 norm as

$$||u||_{H^1(\Omega)} = \left(||u||^2_{L_2(\Omega)} + |u|^2_{H^1(\Omega)} \right)^{\frac{1}{2}}. \tag{2.4}$$

To find solutions where $u = g$ on $\partial\Omega_D$, solutions are here sought in a subspace of $H^1(\Omega)$ into which these boundary values have been submitted. This space is defined as

$$V_g = \{v \in H^1(\Omega) : v = g \text{ on } \partial\Omega\}. \tag{2.5}$$

Notation

An often used notation for (2.3) is

$$\begin{aligned} a(u, v) &= (\nabla u, \nabla v)_\Omega - (\nabla u \cdot \mathbf{n}, v)_{\partial\Omega}, \\ L(v) &= (f, v)_\Omega, \end{aligned}$$

where for a set W, $(\cdot, \cdot)_W$ denotes the $L_2(W)$ scalar product. Using this notation, the abstract problem

$$a(u, v) = L(v) \quad v \in V_g. \tag{2.6}$$

is formulated. Here, $a(u, v)$ is referred to as the *bilinear form*, which contains all terms with both the *trial function* u and the test function v. $L(v)$ is the *linear form*, containing all terms without the trial function.

2.1.2 Discrete variational formulation

For a numerical approach, an infinite dimensional space such as V_g is not feasible. Considering instead a finite dimensional subspace of V_g, a numerical approximation to the solution can be found by searching for an approximate solution to a *discretized* variational formulation.

The continuous variational formulation (2.3) is converted into a discrete problem using a finite dimensional subspace of our infinite dimensional space V_g. A discrete problem is thus finding $u_h \in V_g^h \subset V_g$ such that for all $v_h \in V_g^h$:

$$a_h(u_h, v_h) = L_h(v_h) \tag{2.7}$$

The above equation, known as the *Galerkin equation*, allows for the computation of u_h numerically as a finite linear combination of the basis vectors of V_g^h. A solution is thus sought in a function space where by an ansatz the *trial function* u_h can be expressed with basis functions $\{\varphi_i(x)\}_{i=1}^N \subset V_g^h$ as

$$u_h(x) = \sum_j^N c_j \varphi_j(x) \tag{2.8}$$

where the coefficients $\{c_j\}_{j=0}^N$ are unknowns to be computed. Each unknown c_j is sometimes called a *degree of freedom* (DOF), referencing the role it can be thought to play in *configuring a physical system.* The ansatz (2.8) leads to the linear system

$$\sum_{j \in \mathcal{I}_V} A_{i,j} c_j = b_i \quad i \in \mathcal{I}_V$$

where, for the Poisson's equation (2.1),

$$A_{i,j} = \int_\Omega \nabla\varphi_i \nabla\varphi_j \, \mathrm{d}\mathbf{x} \; = a(\varphi_i, \varphi_j), \quad b_i = \int_\Omega f_i \varphi_i \, \mathrm{d}\mathbf{x} + \int_{\partial\Omega} g_{N\,i} \varphi_i \mathrm{d}s = L(\varphi_i). \tag{2.9}$$

To successfully assemble this linear system, what remains is to construct the space $V_g^h \subset V_g$ wherein a solution is sought. The construction of this function space is done in two parts: first the domain Ω is decomposed into a discrete computational domain, then, upon this domain, the function space V_g^h is built.

2.1.3 Meshes

A common discretization $\mathcal{T}$ of the domain Ω is to partition it into a set of disjoint *cells* T:

$$\Omega = \mathcal{T} = \bigcup_{i=1}^N T_i$$

known as a *mesh.* The treatment of meshes is here limited to the real plane $\mathbb{R}^2$. A formal definition of a *triangulation* in this regard is:

Definition 1 (Triangulation)**.** *Let $\Omega \subset \mathbb{R}^2$ be a bounded two-dimensional domain with a polygonal boundary $\partial\Omega$. A* triangulation *$\mathcal{T}$ of Ω is a set $\{T\}$ of triangles T, such that $\Omega = \bigcup_{T \in \mathcal{T}}$ where no triangles intersect, except at edges or vertices. No vertices are allowed to be* hanging*, that is, lie on the edge of another triangle.*

An example triangulation of a square is given in in Figure 2.1. A quantity belonging to a triangulation is the *local mesh size*, h_T, which is a size measure of a triangle in the mesh, denoting the length of the longest edge or the circumradius of a triangle. The *mesh size*, here denoted h, is the maximum local mesh size in the global triangulation. Further terminology associated with a triangulation is that of *degeneracy*; a triangulation is said to be degenerate if two vertices in a triangle in the domain share the same coordinate, or if the vertices all lie on a line.

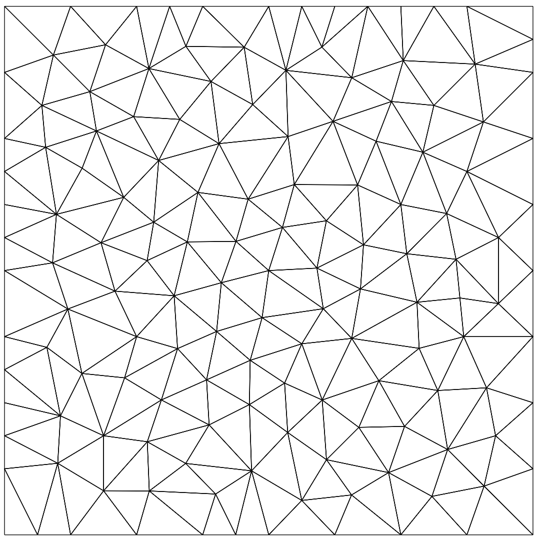


Figure 2.1: Unstructured mesh of a unit square.

2.1.4 Piecewise polynomial spaces

With a triangulation $\mathcal{T}$ it is possible to construct a function space where problems such as (2.7) can be solved. One such space is the space of all continuous piecewise linear polynomials over $\mathcal{T}$, which is here derived by first making a function space on each individual triangle in $\mathcal{T}$ and then putting these together to form a function space for the whole domain.

Let T be a triangle and let $P_1(T)$ be the space of linear functions on T, defined by

$$P_1(T) = \{v : v = c_0 + c_1x_1 + c_2x_2,\ (x_1, x_2) \in T,\ c_0, c_1, c_2 \in \mathbb{R}\}$$

and let N_r for $r = 1, 2, 3$ denote three *local nodes* in the triangle T, typically coinciding with the vertices of T. Any function v in the space $P_1(T)$ is now shown to be uniquely defined by what is called the *nodal values* $\alpha_r = v(N_r)$. This follows by computing the determinant of the linear system

$$\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{bmatrix} \tag{2.10}$$

and finding that its absolute value equals $2|T|$, where $|T|$ is the area of T. If this area is positive then the linear system has a non-zero determinant, implying that the linear system has a unique solution as long as the triangle T is not degenerate. Thus any function in the space $P_1(T)$ is uniquely described by the nodes $N_i = (x_1^i, x_2^i)$ and a nodal value associated with each of these. To use the nodal values $\alpha_r = v(N_r)$ as the degrees of freedom in (2.8), a *nodal basis* $\{\lambda_1, \lambda_2, \lambda_3\}$ is introduced, replacing the natural basis $\{1, x_1, x_2\}$ for $P_1(T)$. The nodal basis is defined by

$$\lambda_s(N_r) = \begin{cases} 1, & r = s \\ 0, & r \neq s \end{cases}, \quad r, s = 1, 2, 3 \tag{2.11}$$

Using this new basis, any function v in $P_1(T)$ can be expressed as

$$v = \alpha_1\lambda_1 + \alpha_2\lambda_2 + \alpha_3\lambda_3$$

where $\alpha_r = v(N_r)$.

Now, to construct the finite dimensional subspace V_g^h on the chosen discretization of the domain $\mathcal{T} = \{T\}$, a first requirement is that for each triangle T, any function v in this space belongs to $P_1(T)$. A second requirement is continuity between neighboring triangles. By looking at the shared edge $E = T_1 \bigcap T_2$ of two neighbouring triangles in the discretization $\mathcal{T}$, it's evident that $v_1 = v_2$ on E, since the linear polynomials $v_1 \in P_1(T_1)$ and $v_2 \in P_1(T_2)$ coincide at the endpoints. Lastly, interpolating the Dirichlet condition g along the nodes N_j belonging to the boundary $\partial\Omega$ with a suitable interpolant $I_{h,\partial\mathcal{T}}$ and submitting these values into the space, the space of all continuous piecewise linear polynomials conforming to g on the boundary is obtained:

$$V_g^h = \{v_h \in C^0(\Omega) : v_h|_T \in P_1(T), \forall T \in \mathcal{T} \text{ and } v_h = I_{h,\partial\mathcal{T}}g \text{ on } \partial\Omega\} \tag{2.12}$$

where $C^0(\Omega)$ is the space of all continuous functions on Ω.

Analogous to the nodal basis (2.11), a global basis $\{\varphi_j\}_{j=1}^{n_p} \subset V_g^h$ is defined

$$\varphi_j(N_i) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \quad i, j = 1, 2, ..., n_p \tag{2.13}$$

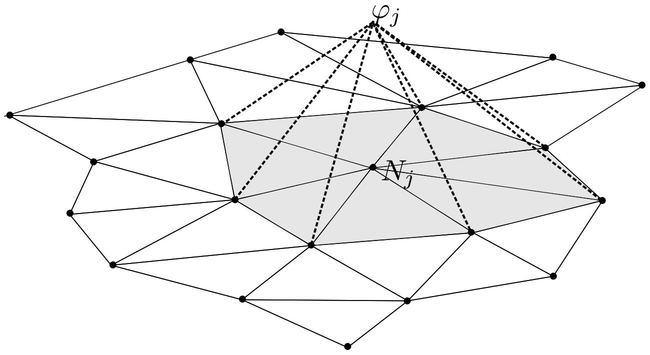


Figure 2.2: The global basis $\{\varphi_j\}_{j=1}^{n_p}$ over $\mathcal{T}$ shown at node N_j.

Using this basis any function $v_h \in V_g^h$ can be expressed as

$$v_h = \sum_{j}^{n_p} \alpha_i\varphi_i$$

where $\alpha_i = v(N_i), i = 1, 2, ..., n_p$.

When submitting the boundary values g into the function space V_g^h like above, the boundary condition is said to be *strongly enforced.* A suitable the linear interpolant $I_{h,\partial\mathcal{T}}$ of a continuous function g on the boundary $\partial\mathcal{T}$ is formed by using the global basis above as

$$I_{h,\partial\mathcal{T}}g = \sum_{i \in \mathcal{I}_{h,\partial\mathcal{T}}} g(N_i)\varphi_i$$

where $\mathcal{I}_{\partial\mathcal{T}}$ is an index-set running along the boundary $\partial\mathcal{T}$. A similar interpolant can be defined for a function f over a triangle T is

$$I_{h,T}f = \sum_{r=1}^{3} f(N_r)\varphi_r \tag{2.14}$$

This interpolant defines a plane running through the three nodes N_r, and is used below in error estimates.

The finite element

The function space V_g^h based on the linear polynomials P_1 over a triangle T is only one of many function spaces suitable for the discrete variational formulation (2.7). Let the set of nodal values over T be $\mathcal{N} = \{\alpha_1, \alpha_2, \alpha_3\}$. The triple $(T, \mathcal{N}, P_1(T))$ is commonly called a *finite element*, and the specific element shown above is the The *Lagrange element* of order 1, here simply referred to as P_1 elements.

2.1.5 Assembly and numerical integration

To construct the linear system (2.9), the integrals in (2.9) have to be evaluated. In the P_1 is done by integrating over the local function spaces $P_1(T)$ and adding these contributions to the global linear system.

Assembly

For simplicity it is here assumed that the location of vertices coincide with the location of the degrees of freedom in the computational domain, as is typically the case with P_1 triangular elements. Let r and s be local node numbers in a triangle T, and let the corresponding global node numbers i and j be obtained through the mappings $i = q(T, r)$ and $j = q(T, s)$. Algorithm 1 shows how the global matrix A is assembled using these mappings. The vector b_i for $i = 1, 2, ..., N$ in (2.9) is assembled in a similar manner.

Algorithm 1 Assembly of global matrix

```
$A = 0$
for each $T \in \mathcal{T}$ :
    $\mathcal{I}(T) = \{1, ..., dim(P_1(T))\} \times \{1, ..., dim(P_1(T))\}$
    for each $(r, s) \in \mathcal{I}(T)$ :
        $A^{(T)}_{r,s} = a(\varphi^{(T)}_r, \varphi^{(T)}_s)$
    for each $(r, s) \in \mathcal{I}(T)$ :
        $A_{q(T,s),q(T,r)} += A^{(T)}_{r,s}$
```

Numerical integration

Numerical integration, or *quadrature*, is calculating the definite integrals as they appear in for instance (2.9), be it exactly or approximatively, by a numerical rule. A general quadrature rule on a triangle T takes the form

$$\int_T f \, \mathrm{d}x \approx \sum_j w_j f(q_j)$$

where the set $\{q_j\}$ are *quadrature points* distributed within T and $\{w_j\}$ the corresponding *quadrature weights*.

Choosing a quadrature rule that can integrate polynomials exactly on the chosen function space is usually advantageous. For continuous piecewise linear functions such a rule is the two dimensional analogue to the Trapezoidal rule, the *corner*-formula:

$$\int_K f \, \mathrm{d}x \approx \sum_j \frac{|T|}{3} f(N_j).$$

where N_j denotes the corners of triangle T. For Lagrange elements of order 2, P_2, the *Simpson's rule* can be used to integrate the cubic polynomials exactly.

2.2 Errors

A typical error estimate in the L_2-norm between an analytical solution u and a numerical solution u_h of the Poisson's equation when using P_1 elements is

$$||u - u_h||_{L^2(\Omega)} \leq Ch^2 ||D^2 u||_{L^2(\Omega)}, \tag{2.15}$$

where h is a mesh size and where D^2 denotes the total first and second derivatives of u, that is $D^2 u = \sqrt{|\partial^2 u/\partial x^2|^2 + |\partial^2 u/(\partial x^2 \partial y^2)|^2 + |\partial^2 u/\partial xy^2|^2}$. This error shows that the numerical solution is converging to the analytical solution in L_2 norm quadratically as you decrease the mesh size h. For a rigorous analytical treatment of various errors associated with numerical approximations of PDEs like the one above, consult Scott and Brenner [24] or Larson and Bengzon [25], where proofs of the above a priori error estimate can be found. The treatment of errors is in what follows limited to cell shape, expressed by edge lengths in a triangle.

Approximating the solution to the model problem (2.1) can in principle be performed on any mesh conforming to Definition 1 above, but to limit potential sources of error in the approximation, a proper domain discretization has to meet several requirements. The shape and size of the *cells* in a mesh, be it triangles or tetrahedrons, affects both the properties of the numerical solution of our problem; i.e. how accurate the approximation is, and how the solution is obtained; i.e. how difficult a linear system is to solve. To this end various metrics exist to measure the *quality* of a triangulation. A commonly used metric is the *chunkiness* parameter k_T: Let d_T be the diameter of the inscribed circle in triangle T, then

$$k_T = \frac{h_T}{d_T}$$

Definition 2 (Shape regularity)**.** *A triangulation $\mathcal{T}$ is said to be* shape regular *if there is a constant $k_0 > 0$ such that*

$$k_T \geq k_0, \quad \forall T \in \mathcal{T} \tag{2.16}$$

This parameter is a general indicator of how well a mesh performs with respect to numerical errors and matrix conditioning for different problems.

2.2.1 Cell shape and numerical errors

For a brief exposition of how mesh size and shape can affect a numerical approximation, assume $I_{h,T}f$ (2.14) is a linear approximation of a smooth function f over a single element T. Schewchuk [26] derives error bounds for the *interpolation error*: $f - I_{h,T}f$, and for the error in the gradient of the approximation: $\nabla f - \nabla I_{h,T}f$, a property that in many applications is just as important as the interpolation error itself. Specific weak and simple upper bounds for these errors as presented in [26] are

$$||f - I_{h,T}f||_\infty \leq c_T \frac{\ell_{\max}^2}{3}$$
$$||\nabla f - \nabla I_{h,T}f||_\infty \leq c_T \frac{3\ell_{\max}\ell_{\text{med}}\ell_{\min}}{2A}$$

where the *sup*-norm $||f||_\infty = sup\{|f(x)| : x \in T$ is used and where $\ell_{\max}$, ℓ_{med} and $\ell_{\min}$ denotes the length of longest, median and minimum edge in the triangle and c_T denotes a upper bound on the *curvature* of f bounded to the particular triangle T. See [26] for tighter, more complex bounds and a precise definition of the curvature c_T of f.

The bound on the interpolation error shows that this error is largely independent of shape; it is simply bounded by the size of the triangulation (the longest edge in the triangle) and how much the function f locally varies over the domain. The error in the approximation of the gradient, on the other hand, is heavily influenced by large angles in the triangulation, and explodes as the *product of edge lengths/area*-ratio becomes large, as illustrated in Figure 2.3.

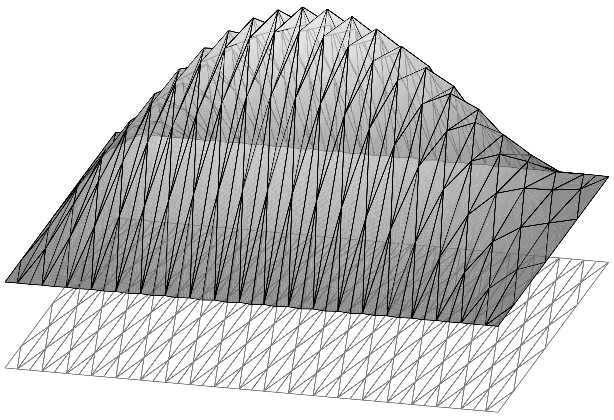


Figure 2.3: Large angles in a mesh resulting in poor estimation of $||\nabla u - \nabla g||_\infty$ for the model problem (2.1). Generation and problem code can be found in Appendix A.4.

2.2.2 Cell shape and matrix conditioning

The shape and difference in size of elements in a mesh also affect the conditioning of the linear system resulting from a discretization of a problem. Problems involving the Laplace term Δu gives rise to the matrix $A_{i,j}$ in (2.9), known as a *stiffness matrix*, here denoted K. A common way to assess the solubility of a matrix is by defining the condition number:

$$\kappa = \lambda^K_{\text{max}} / \lambda^K_{\text{min}} \tag{2.17}$$

where λ^K_{max} and λ^K_{min} is the largest and smallest eigenvalue of K. A linear system with a large condition number is known to be sensitive to numerical errors.

In the finite element method the assembled global linear system is composed of contributions from single element matrices, like shown in Algorithm 1. In this light, Fried [27] shows that λ^K_{min} is proportional to the area of the smallest element, and also that the largest eigenvalue λ^K_{max} of the global stiffness matrix satisfies

$$\max_T \lambda^T_{\text{max}} \leq \lambda^K_{\text{max}} \leq m \max_T \lambda^T_{\text{max}} \tag{2.18}$$

where λ^T_{max} denotes the largest eigenvalue of the element stiffness matrices and m is the maximum numbers of elements meeting at a single vertex. Further it can be shown [26] that the eigenvalues of the element stiffness matrix for a P1 element using exact integration is

$$\lambda^T = \frac{\ell^2_{\text{max}} + \ell^2_{\text{med}} + \ell^2_{\text{min}} \pm \sqrt{(\ell^2_{\text{max}} + \ell^2_{\text{med}} + \ell^2_{\text{min}})^2 - 48A^2}}{8A} \tag{2.19}$$

Looking at (2.19) in conjunction with (2.18) and (2.17) it is evident that a single badly shaped element in a mesh can ruin the conditioning of the whole global linear system. A properly meshed domain thus prescribes elements where the *sum of edge lengths/area*-ratio is satisfying for all elements, i.e. prescribing elements where no angles are too narrow or too wide.

2.2.3 Meshing domains of viable quality

To produce meshes that satisfy the above requirements various construction and refinement techniques are used. *Structured* meshes, consisting of cells more or less homogeneously shaped, aligned and distributed around a domain, are often used in theoretical applications and in structural engineering where the object analyzed can be represented by a simple geometrical shape like a rectangle or triangle can be put into lattice arrangements. Unstructured meshes, like shown in Figure 2.1, are common when solving problems on geometric data obtained from medical scans or complex geometrical models constructed using CAD software. To this end algorithms producing *Delanuay triangulations* based on these geometrical descriptions are often employed. A Delanuay triangulation for a set of points P is a triangulation $\mathcal{T}_P$ such that no point $p \in P$ is inside the circumcircle of a triangle $T \in \mathcal{T}_P$. If the points P are spread more or less *evenly* within the convex hull containing P, this triangulation tends to avoid triangles with small or large angles. To produce a triangulation conforming to a domain Ω with triangles as ideal as possible, various algorithms with various point-placement strategies can be used [28, 29]. Delanuay triangulations are by far the most popular approach to construct unstructured meshes, but no algorithm has been found to automatically mesh domains, given any input, in an universally satisfying manner. Because of this, human interference in the meshing process is common, and a wide range of meshing tools for specific applications exist. General open source triangulation tools often used are TetGen [30], Gmsh [31], Netgen [32] and Triangle [33]. For biomedical applications tools such as VMTK [34] and Pyformex [35] are commonly used.

Chapter 3

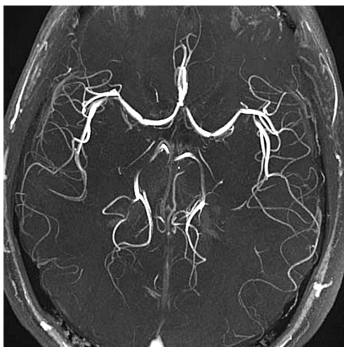
The finite element method on unfitted geometries

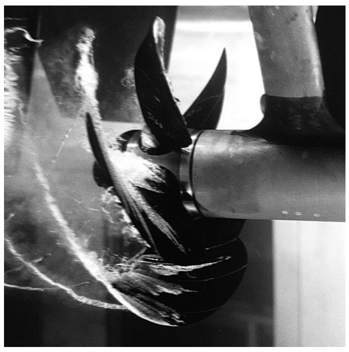
As mentioned in the introduction, many existing applications of the finite element method can benefit from a more flexible geometrical approach than the one permitted by a standard mesh discretization as presented in Section 2.1.3. Meshing complex domains to comply to the requirements of the standard FEM is a time consuming affair, often limiting the geometric configurations explored in a given application and also prohibiting many ways to automatically set up and reconfigure the domain. Many applications of FEM is further limited by the standard mesh discretization when the computational domain undergoes deformations or when geometrical features of interest are hard to represent.

In fluid-structure interaction (FSI) problems, where the computational domain potentially undergoes large deformations, describing the surface by an independent mesh or a function defining a surface alleviates many concerns [36]. Similar geometrical descriptions would be beneficial in two-phase flow problems [37, 5], where the interface between the two fluids are traditionally described by a tessellated surface corresponding to vertices within a single mesh.

When domains are constructed from data returned from medical scans, like that of an magnetic resonance imagery (MRI) or X-ray computed tomography (CT), meshing domains to acceptable quality is often a complex and error prone task [38]. In addition, some features, like blood vessels (Figure 3.1a) or neurons are not easily represented at all. For many medical applications, a flexible geometrical approach allowing representations of these physical features as networks of intervals, illustrated below in Figure 3.2a, can be beneficial. An application of this is shown in Cattaneo and Zunino [9], where coupling a network of intervals with a 3D surrounding is used to model tissue perfusion and microcirculation.

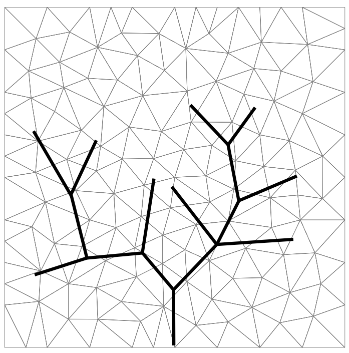
Data returned from scans such as above and other 3D imaging techniques are typically *voxels* describing some property of the scanned material, like its density, or a mapping of the diffusion of particular molecules (diffusion tensor MRI). A geometrical approach capable of setting up a computational domain directly from such voxel data, like that associated with *fictitious domain methods* [12, 3, 39], one of which is illustrated in Figure 3.2c, is naturally practical.


(a) Time-of-flight MRA at the level of the Circle of Willis in the human brain.

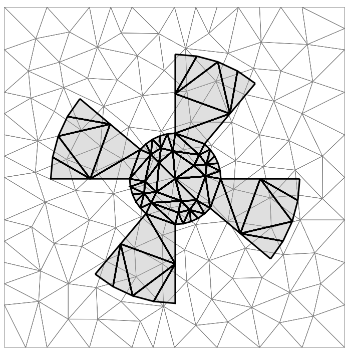

(b) Rotating propeller forming vapor cavities in water.

(c) Non-Newtonian fluid (corn starch and water) dancing on a loudspeaker. Photo by Rory MacLeod, CC BY 2.0.

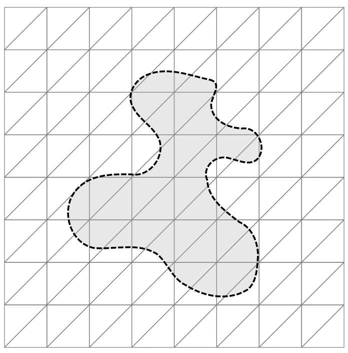
Figure 3.1: Physical phenomena benefiting from flexible geometrical approaches to numerical simulation. **(a)** Blood vessels like veins or capillaries are hard to represent in standard mesh discretizations. **(b)** Physical configurations like that of a propeller connected to a motor are often meshed as static in many discretizations. **(c)** Surfaces of fluids undergoing large deformations over time are hard to represent in a typical mesh.



(a) Network of intervals superimposed on a unstructured background mesh in 2D.



(b) Overlapping meshes, one mesh superimposed on another, together forming full domain.



(c) Fictitious domain, boundary of domain defined by a function and submerged in a structured background mesh.

Figure 3.2: Flexible geometrical configurations applicable to physical phenomena shown in Figure 3.1.

3.1 Solving partial differential equations on unfitted geometries

In Section 2.1.2 solutions u were sought in the function space V_g^h (2.12) where the boundary conditions $u = g$ on $\partial\Omega$ where incorporated into this function space itself. To achieve this, g is typically interpolated along the boundary and submitted as pointwise values into V_g^h. As Hansbo [16] elegantly states, this *goes against the grain of the main idea of the finite element method as a weak method that only fulfills equations in the mean.* This principle does not cause any major trouble for boundary value problems on typical triangulations, but when moving on to more general interface problems, the same ideas are often hard to employ.

In the case of *unfitted* geometries like shown in Figure 3.2, it is not possible to submit pointwise values of boundary conditions into the function space or prescribing interface conditions at some DOFs as it is typically done in the case where the geometry is fitted. The DOFs are no longer located at the boundary or interface describing a surface in the mesh and hence a different mechanism is needed to impose these boundary and interface conditions.

Nitsche's method and the other techniques mentioned in the introduction all rely on suitable domain decompositions. A large part of this thesis is devoted to surveying and implementing algorithms to compute such domains decompositions and the presentation of the Nitsche method is here thus primarily concerned with exposing the various geometrical entities it requires to be applied successfully. Thorough mathematical analysis of this method for different problems can be found in Massing [12], Hansbo [16], Sogn [40]. The presentation of the Nitsche method is in notation and structure closely related to that of Massing [12, 13] and Hansbo [16].

As an application of the Nitsche method on overlapping meshes, *domain bridging* is at the end of this chapter formulated between two domains. This geometrically flexible approach allow a problem to be solved on two or more meshes, and is a useful technique to avoid many of the challenges related to meshing as described in Section 2.1.3. Furthermore, the computational domain decomposition tools needed for doing this task also covers most of the machinery needed for the other cases shown in Figure 3.2.

3.2 The Nitsche method

The Nitsche method [11] is a way of weakly imposing boundary conditions and interface conditions. It is here presented by means of example. In the first example Dirichlet boundary conditions are transformed into weak terms in the variational formulation, in a way similar to the formulation of a Neumann boundary condition. The numerical solution u_h is then sought in a function space that does not have any apriori knowledge of the boundary data. In the second example the Nitsche method is formulated to weakly enforce interface conditions on fitted geometries, and in the last example, this formulation is carried over to the case of unfitted geometries.

3.2.1 Weakly enforcing boundary conditions by the Nitsche method

Starting with the continuous variational formulation of the Poisson's equation (2.3) in operator notation

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

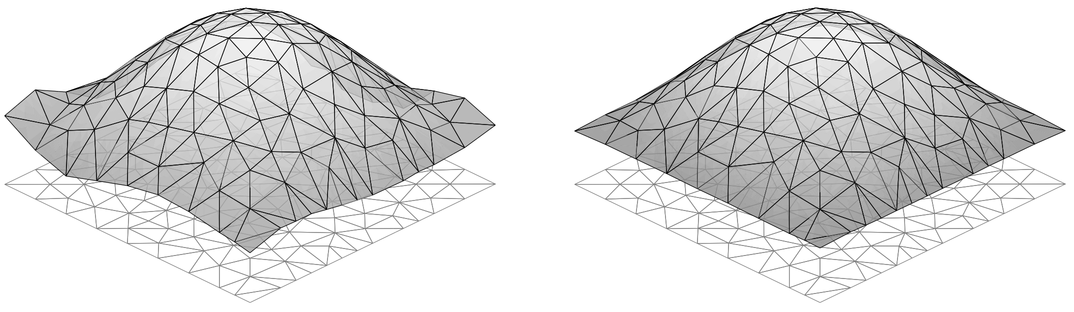
the idea of Nitsche's method is to weakly enforce the boundary condition by penalizing the jump $u - g$ between the unknown solution u and the boundary condition g in the discretized variational formulation:

$$(\nabla u_h, \nabla v_h) - (\nabla u_h \cdot \mathbf{n}, v_h)_{\partial\Omega} + \gamma(h^{-1}(u_h - g), v_h)_{\partial\Omega} = (f, v_h)_\Omega$$

where $\gamma > 0$ a penalty parameter and h is the piecewise constant mesh size function defined by $h|_T = h_T$ with h_T being the diameter of an element $T \in \mathcal{T}_h$. For $\gamma > \gamma_0$, Nitsche [11] proves that this gives an optimally convergent method. With the added term, this system lacks symmetry. Symmetry is important for many linear system solution algorithms, and is recovered by adding an additional term $(\nabla v \cdot \mathbf{n}, (u - g))_{\partial\Omega}$ to the problem [12, 16]. All in all the Nitsche method for this problem is: find $u_h \in V^h$ such that $\forall v_h \in V^h$;

$$\begin{aligned}(\nabla u_h, \nabla v_h)_\Omega - (\nabla u_h \cdot \mathbf{n}, v_h)_{\partial\Omega} - (\nabla v_h \cdot \mathbf{n}, u_h)_{\partial\Omega} + \gamma(h^{-1} u_h, v_h)_{\partial\Omega}& \\ = (f, v_h)_\Omega - (\nabla v_h \cdot \mathbf{n}, g)_{\partial\Omega} + \gamma(h^{-1} g, v_h)_{\partial\Omega}&\end{aligned}$$

where V^h is similar to the function space V_g^h (2.12) without the submitted boundary values. More on this method for boundary values can be found in [16, 17]. Figure 3.3 shows the Poisson's equation solved with the Nitsche method for two γ values.



(a) Oscillations at border, $\gamma = 1$. **(b)** $\gamma = 10$.

Figure 3.3: Two solutions of the Nitsche method applied to the Poisson problem with varying γ parameters and $f(x, y) = 2\pi^2(\sin(\pi x)\sin(\pi y))$. For γ of insufficient magnitude, $\gamma = 1$, remnants of the source function f is seen at the boundary. The source code for this can be found in the appendix, A.1.

3.3 Weakly enforcing interface conditions by the Nitsche method

Now a finite element method for solving the Poisson equation on a domain Ω consisting of two subdomains Ω_1 and Ω_2 separated by the interface $\Gamma = \partial\Omega_1 \cap \partial\Omega_2$ is derived. The test function $v : \Omega \to \mathbb{R}$ can then be thought of as a composition $v = (v_1, v_2)$ where $v_i = v|_{\Omega_i}, i = 1, 2$ is the restriction of v to each domain. For simplicity it is assumed that $\overline{\Omega_2} \subseteq \Omega_1$ as shown in Figure 3.4. A Poisson's equation formulated for the two domains is then: find u_1 and u_2 such that

$$\begin{aligned} -\Delta u_1 &= f_1 && \text{in } \Omega_1, \\ -\Delta u_2 &= f_2 && \text{in } \Omega_2, \\ [\nabla u \cdot \mathbf{n}] &= 0 && \text{on } \Gamma, \\ [u] &= 0 && \text{on } \Gamma, \\ u &= 0 && \text{on } \partial\Omega_{0,D}, \\ \nabla u \cdot \mathbf{n} &= 0 && \text{on } \partial\Omega_{0,N} \end{aligned} \tag{3.1}$$

where $\mathbf{n}$ is the outward pointing unit normal of Ω_1 and $[u] = u_1 - u_2$ denotes a *jump* over the interface Γ. This is referred to as the *interface condition.*

Discretizing the domain Ω_1 and Ω_2 so that the computational domains are *fitted* can be done as shown in Figure 3.4, where the cells in meshes $\mathcal{T}_1$ and $\mathcal{T}_2$ correspond at vertices along the interface Γ.

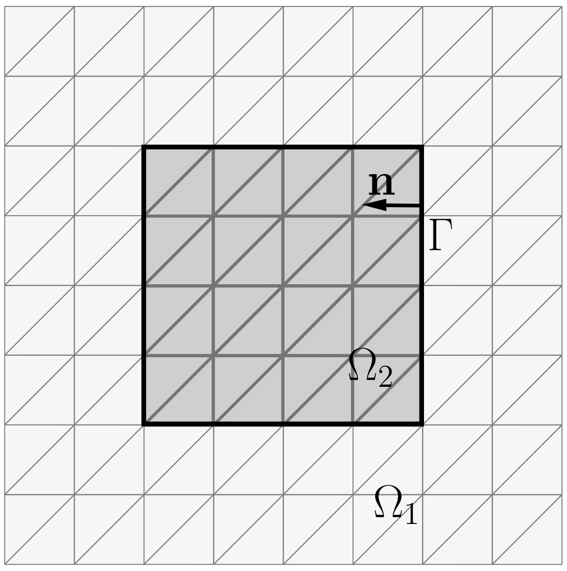


Figure 3.4: Interface Γ between domains, consisting of triangulations of domains Ω_1 and Ω_2 that at the interface Γ coincide at vertices.

The decomposition of Ω in this way means introducing the finite dimensional function space

$$V_g^h = V_0^h \oplus V_1^h, \tag{3.2}$$

the *direct sum* of V_0^h, belonging to the domain $\Omega_1 \cup \Gamma$, and V_1^h, belonging to the domain Ω_2. The finite element functions are then on the form $v_h = (v_h^0, v_h^1)$. Defined on (3.2) is the norm

$$||v||^2_{H_1(\Omega_1 \cup \Omega_2)} = ||v^0||^2_{H_1(\Omega_1)} + ||v^1||^2_{H_1(\Omega_2)}.$$

With $\mathcal{T}_1$, $\mathcal{T}_2$ and the function space (3.2), the strong continuity conditions $[\nabla u \cdot \mathbf{n}] = 0$ on Γ and $[u] = 0$ on Γ are replaced by terms in the discrete variational formulation. The problem can then be stated:

$$a(u_h, v_h) = l(v_h)$$

where

$$a(u_h, v_h) = (\nabla u_h, \nabla v_h)_\Omega - \overbrace{(\langle \nabla u_h \cdot \mathbf{n}\rangle, [v_h])_\Gamma}^{\text{Consistency}} - \overbrace{([u_h], \langle \nabla v_h \cdot \mathbf{n}\rangle)_\Gamma}^{\text{Symmetry}} + \overbrace{(\gamma h^{-1}[u_h], [v_h])_\Gamma}^{\text{Penalty}}, \tag{3.3}$$

$$l(v_h) = (f_h, v_h)_\Omega. \tag{3.4}$$

Here the notation $(\cdot,\cdot)_\Omega = (\cdot,\cdot)_{\Omega_1} + (\cdot,\cdot)_{\Omega_2}$ is used, and

$$\langle \nabla u_h \cdot \mathbf{n}\rangle = \alpha_1 \nabla u_1 \cdot \mathbf{n} + \alpha_2 \nabla u_2 \cdot \mathbf{n} \tag{3.5}$$

where all convex combinations $\alpha_1 + \alpha_2 = 1$ lead to a consistent method [16]. Typically $\alpha_1 = \alpha_2 = \frac{1}{2}$. A strong enforcement of boundary conditions is assumed.

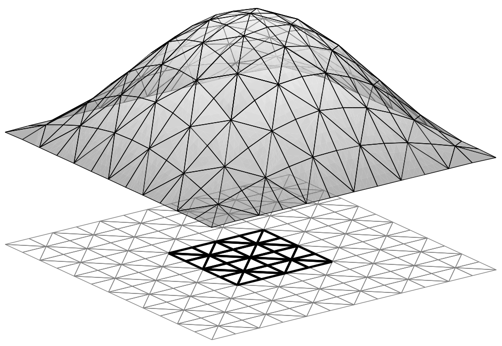


Figure 3.5: Solution of Poisson's equation on fitted composite geometry, thin triangles denote $\mathcal{T}_1$ and thick triangles $\mathcal{T}_2$.

3.4 Overlapping meshes

Superimposing a mesh on another mesh is here referred to as *overlapping meshes.*

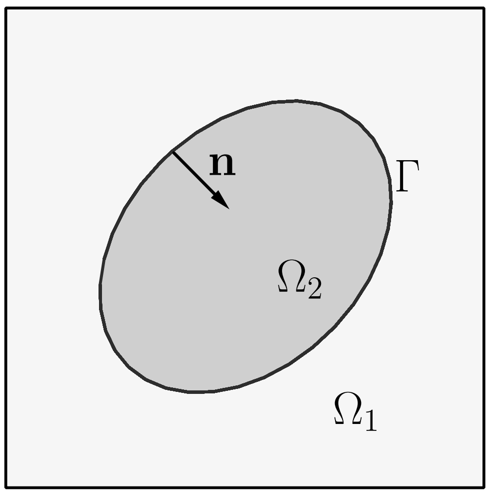


Figure 3.6: The physical domains Ω_1 and Ω_2

Adopting the notation as in Massing [12]; the background mesh $\mathcal{T}_0$ is given for $\Omega = (\overline{\Omega_1 \cup \Omega_2})^\circ$ and another mesh $\mathcal{T}_2$ is given for the overlapping domain Ω_2. The mesh $\mathcal{T}_0$ is decomposed into three disjoint subsets

$$\mathcal{T}_0 = \mathcal{T}_{0,1} \cup \mathcal{T}_{0,2} \cup \mathcal{T}_{0,\Gamma}$$

where

$\mathcal{T}_{0,1} = \{T \in \mathcal{T}_0 : T \subset \overline{\Omega}_1\}$	is the set of **not** overlapped elements relative to Ω_2,
$\mathcal{T}_{0,2} = \{T \in \mathcal{T}_0 : T \subset \overline{\Omega}_2\}$	is the set of **completely** overlapped elements relative to Ω_2,
$\mathcal{T}_{0,\Gamma} = \{T \in \mathcal{T}_0 : T \cap \lvert(\Omega_1 \cup \Omega_2)\rvert > 0\}$	is the set of **partially** overlapped elements relative to Ω_2.

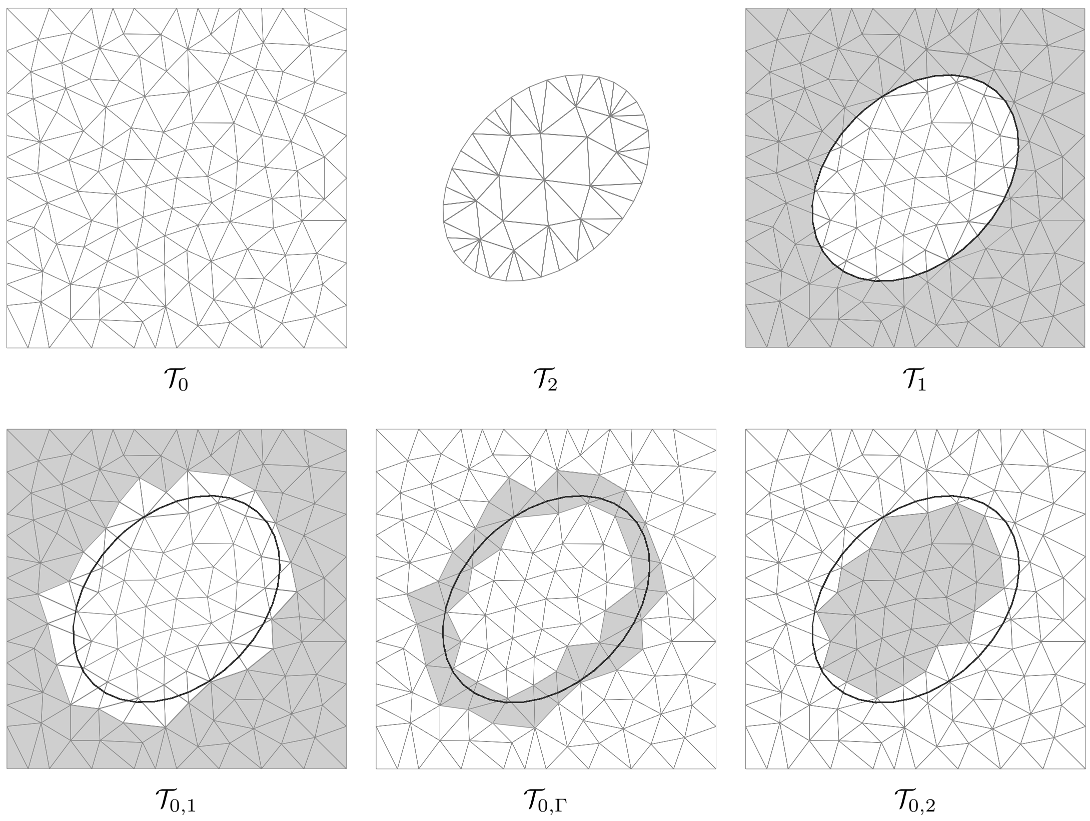


Figure 3.7: Notation for domains. $\mathcal{T}_0$ is referred to as *background mesh*, $\mathcal{T}_2$ as the *overlapping mesh.* $\mathcal{T}_1 \bigcup \mathcal{T}_2$ is the combined computational domain. $\mathcal{T}_{0,1}$, $\mathcal{T}_{0,\Gamma}$ and $\mathcal{T}_{0,2}$ are decompositions of the background mesh with respect to the interface Γ between $\mathcal{T}_0$ and $\mathcal{T}_2$.

The reduced and physical part of background mesh $\mathcal{T}_0$ corresponding to Ω_1 is defined by

$$\begin{aligned} \mathcal{T}_1^* &= \mathcal{T}_{0,1} \cup \mathcal{T}_{0,\Gamma}, \\ \mathcal{T}_1 &= \{T \cap \overline{\Omega}_1 : T \in \mathcal{T}_1^*\}, \end{aligned}$$

respectively. Shape-regularity (2.16) is required on both meshes $\mathcal{T}_0$ and $\mathcal{T}_2$. Further it is assumed that the mesh sizes are *compatible* over the interface Γ, that is, there exist a mesh-independent constant $C > 0$ such that for all $T_0 \in \mathcal{T}_0$ and all $T_2 \in \mathcal{T}_2$ the condition

$$C^{-1} h_{T_0} \leq h_{T_2} \leq C h_{T_0} \tag{3.6}$$

is satisfied whenever $T_0 \cap T_2 \cap \Gamma \neq \emptyset$, which essentially means that the cells in each mesh does not differ *wildly* from each other in size over the interface Γ. This is a reasonable assumption for now, since is does not make very much sense to have a very large number of degrees of freedom in one mesh being *configured* by only few in the other.

The relevant function spaces for this problem is defined on $\mathcal{T}_1^*$ as

$$V_1^h = \left\{ v_h|_{\Omega_1} : v_h \in V_1^h(\mathcal{T}_1^*) \right\} \tag{3.7}$$

and on $\mathcal{T}_2$ a standard function space. In the P_1 case, V_1^h and V_2^h are equivalent to (2.12) over $\mathcal{T}_1^*$ and $\mathcal{T}_2$, respectively with and without submitted boundary values (if choosing to strongly enforce these.) The function space used for the unfitted geometry thus is

$$V_g^h = V_1^h \oplus V_2^h, \tag{3.8}$$

similar to the function space for the fitted case above.

Now, (3.8) is not the function space on the actual physical domain $\mathcal{T}_1$. In the case of P_1 elements, the basis functions in the this function space, call it $P_1^*(\mathcal{T}_1)$, are illustrated in Figure 3.8.

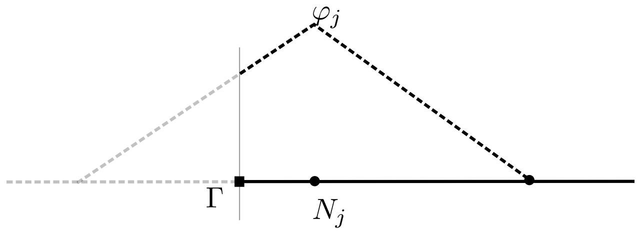


Figure 3.8: *Broken* basis functions in the space $P_1^*(\mathcal{T}_1)$.

In the following a solution of the Poisson's equation over two domains (3.1) will be sought in $P_1^*(\mathcal{T}_1)$, and this is here done by integrating just over the part of the *cut elements* of $\mathcal{T}_{0,\Gamma}$, contained within $\mathcal{T}_1$.

3.5 Challenges

3.5.1 Cut elements

The algorithms needed to decompose multiple meshes into domains containing cut elements suitable for integration requires the use of several techniques from the field of computational geometry. As a prerequisite for any method of integration, information about where the meshes $\mathcal{T}_0$ and $\mathcal{T}_2$ overlap has to be derived. Then, depending on what method is chosen to integrate on these geometries, specific information about the intersection of cells needs to be obtained and an applicable domain decomposition like a triangulation or set of polytopes needs to be constructed and passed on to further numerical machinery. Obtaining this information and then decomposing the domain is a computationally intensive task scaled by the size of the meshes involved. Several methods for efficiently detecting and computing intersections are reviewed, implemented and benchmarked in Chapter 4, *Collision detection and computational domain decomposition.*

3.5.2 Integration and assembly

The standard approach of applying Gaussian quadrature rules (Section 2.1.5) on the cells in a domain is not directly applicable when solving problems on overlapping meshes. The discontinuous nature of the function space (3.2) allows a function to be discontinuous within a single cell, and this is not readily handled by the standard quadrature rules and assembly algorithms. Depending on what kind of decomposition of the domain is available, specific quadrature rules have to be applied and evaluated. Furthermore, the standard assembly process has to be adapted to properly associate the interface with the cut geometry. A selection of ways of integrating on cut elements is reviewed and an assembly algorithm is presented in Chapter 5, *Integration and assembly of cut elements.*

Chapter 4

Collision detection and mesh decomposition

Collision detection is the problem of detecting and describing a collision between two or more objects. As a computational problem this is here split into *if*, and if so, *where* the objects are colliding. Detecting *if*, say, two primitives intersect, typically just requires one condition in an algorithm to terminate, whereas *where* requires a more complete parse of the input geometry.

The first sections in this chapter is devoted to the first condition, *detecting intersections*. In Section 4.1 different techniques are surveyed and considered. Subsequent sections are devoted to two distinct techniques, namely a *bounding volume hierarchy* (Section 4.2) and an *advancing front* algorithm for detecting intersections (Section 4.3), each solving this problem efficiently.

The second problem, precisely determining *where* objects are intersecting is crucial to properly decompose a domain to be used for numerical simulation. Computing local intersections and triangulating the resulting geometry, in ways compatible with integration on cut elements, is covered in Section 4.6: *Computational domain decomposition.*

Alongside descriptions of different techniques, several implementations of associated algorithms are presented and benchmarked on selected geometries typical to the finite element method. Lastly, a data structure for passing the obtained information on to other numerical software is presented in Section 4.6.4.

4.1 Detecting intersections

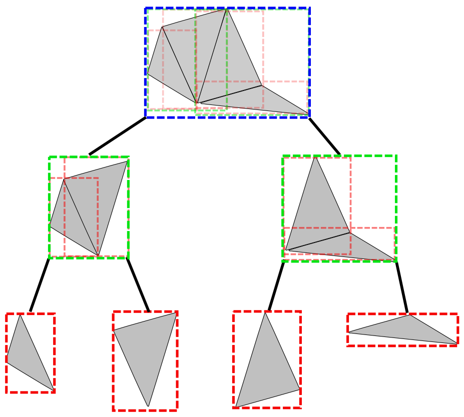
Collision queries such as *where is the first collision between set A and set B?*, *what point in A is closest to point x?* and *how many times does a line b intersect A?*, can all be answered by detecting some kind of intersection between derived or involved data. The task of determining which triangles intersect each other in two meshes $\mathcal{T}_1$ and $\mathcal{T}_2$, can be solved by the naive approach of testing all triangles in $\mathcal{T}_1$ against all triangles in $\mathcal{T}_2$ – an approach of order $\mathcal{O}(n^2)$ if both meshes contain n triangles. When n is large, this computational cost severely affects practical aspects of setting up domains such as those given in Chapter 3. The computational cost of this task can be heavily reduced by using appropriate algorithms.

Many of the techniques investigated here originates from applications such as computer games and design tools. Traditionally, many of these applications rely on a preprocessing step to make collision detection feasible within a *real-time* context. Examples of such real-time contexts are 3D computer games, like iD Software's Quake [41], or 3D modeling suites like Blender [42]. When it comes to overlapping meshes, where meshes potentially contain millions or billions of cells, true real-time capabilities are probably not possible, and arguably not necessary. An implementation should nonetheless respect the fact that it might be a part of a large and complex design or implementation process, warranting both simplicity, modularity and speed.

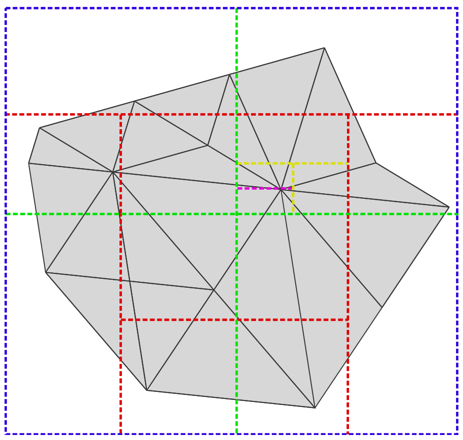
In the scope of treating overlapping meshes within a numerical framework for solving partial differential equations, an implementation of collision detection should be mainly concerned with

- collision queries between simple objects(polygons) and objects of differing dimensionality,
- reconfiguring meshes, either by rotating and translating or by manipulating the mesh vertices,
- swapping data sets completely, refining meshes and changing parts of the geometry,
- performing tests on geometries with large differences in overall shape and structure,
- and having associated storage of predictable size.

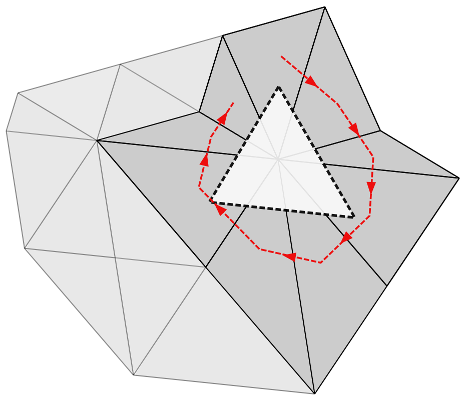
An implementation meeting the above key points can be fulfilled using different techniques, each using various properties associated with the input. Figure 4.1 below gives graphical overview of the ideas behind three more or less distinct techniques.



(a) Object partitioning schemes Objects consecutively wrapped in simple volumes, acting as both a cheap initial *intersection predicate* between primitives and together forming a natural tree structure that can be traversed for collision queries between of these hierarchies.



(b) Spatial partitioning methods Dividing space hierarchically into partitions and finding intersection candidates by location in overall space. Colors denote hierarchical traits similar to those in **(a)**, and the collision query traversal of these hierarchies are very similar.



(c) Advancing front methods White translucent triangle is overlapping gray domain. Detecting all intersections is done by progressively checking if neighbors of a gray triangle already detected to intersect the white triangle, themselves intersect the white triangle, and then moving on to the neighbors of the white triangle in a similar manner. This is done using *mesh connectivity* information.

Figure 4.1: Conceptual illustration of different techniques for detecting intersections between triangular meshes. **(a)** and **(b)** are presented in Section 4.1.1, **(c)** in Section 4.1.2.

4.1.1 Partitioning techniques

In most cases speeding up collision detection relies on exploiting the spatial arrangement of objects. A collection of objects is split into partitions in such a way that it enables a quick determination of which of these partitions participate in a collision or not. Traversing these partitions eventually identify single objects colliding with each other. The most common ways of structuring these partitions can be divided into three categories:

- *Object partitioning schemes* – Construct partitions based on object geometry at all levels. A single object fits in a single partition usually called a *bounding volume.* This method is heavily reliant on the type of bounding volume used, the most common being, axis-align bounding box (AABB) shown in in Figure 4.1a, oriented bounding boxes, sphere and k discrete oriented polytopes (kDOP). The most common name for this technique is a *bounding volume hierarchy.* [22, 20, 43, 44]

- *Spatial partitioning schemes* – Perform a rough pass on geometry, divide overall space into partitions and place objects within these partitions. Objects are allowed to be in several partitions. Common techniques: binary space partitioning (BSP), quadtree shown in Figure 4.1b, kd-trees, octree. [45, 46]

- *Hybrid methods* – Combining the aforementioned methods. Typically using a BVH for partitions containing many objects and switching to i.e. a BSP-tree when a subgroup of five objects or fewer is reached. [22]

All these partitioning methods produce collision queries of one object against a set of n objects of order $\mathcal{O}(\log n)$ when chosen carefully with regard to the underlying geometries and optimized against specific query types - yielding a total complexity of $\mathcal{O}(n \log n)$ when colliding n against n objects [22].

Partitioning schemes are very general concepts, and developing a partitioning scheme for finite element meshes involves carefully selecting needed functionality from several conceptual descriptions and implementations, all intended for a wide range of applications. Many of the aforementioned partitioning algorithms have their origin in game design, which typically deals with a large number of independent objects spread throughout a space. Meshes used for the finite element method usually contain cells that are more or less uniform in shape compared to those in many computer games and distributed throughout a domain in a more homogeneous fashion than is typical here. Additionally, all these cells are usually connected to other cells.

Comparing a object partitioning scheme with spatial partitioning schemes, the main differences are that two or more bounding volumes in a bounding volume hierarchy can cover the same space, and objects are generally only inserted in a single partition, here called a *bounding volume.* In a spatial partitioning scheme the partitions are disjoint and objects contained in the spatial partitioning are typically allowed to be represented in two ore more partitions. The advantage of knowing that the last *leaf node* (the red volumes in Figure 4.1a) in a BVH is an interval, triangle or tetrahedron, makes for easy implementation. Since we are not in interested in the major advantages of spatial partitioning methods, in particularly some kind of *viewport specificity*, that is, easily being able to structure an hierarchy seen from a particular *viewport* in a three dimensional scene, and since both bounding volume hierarchies and spatial partitioning schemes share more or less the same hierarchical traits, spatial partitioning methods are not further investigated in this thesis.

4.1.2 Advancing front methods

A distinction between the techniques shown in Figure 4.1 is whether or not *mesh connectivity* information is used. Mesh connectivity is information derived from a mesh relating a cell with its neighboring cells. Most computational domains used for FEM supply some kind of connectivity information and *advancing front methods* exploits this kind of information to perform collision detection, domain decomposition, or both.

Figure 4.1c illustrates the concept of an advancing front algorithm. Starting from a gray triangle known to intersect the overlapping white triangle, the neighbors of the gray triangle is queued, traversed and tested for intersection with the white triangle. When all the triangles intersecting this particular white triangle are found, the neighbors of this white triangle are again queued and traversed in a similar manner.

Advancing front methods have the advantage of being *optimal* for some tasks, that is, scaling linearly with an input of n objects, $\mathcal{O}(n)$. Such tasks are finding the intersection between two meshes and finding a triangulation of the union between the two [21, 5]. Although optimal for some purposes, on some geometries, they are not easily re-purposed for others, since the geometry of the input and information supplied along with it is tightly connected to the inner workings of the algorithms. Making them work on domains consisting of different primitives, say triangles and tetrahedrons, typically warrants standalone implementations. In Section 4.3 a specific advancing front algorithm is presented and in Section 4.5.6 this and implementation of this is benchmarked and compared to the performance of a BVH.

4.2 The bounding volume hierarchy

Before testing the actual geometry of a complex object wrapping it up in simpler geometry and then testing this geometry first, often yields significant performance gains when detecting intersections on large sets. A *bounding volume hierarchy* (BVH) extends this idea to subgroups of such objects. In an hierarchical fashion, bounding volumes are wrapped on top of other bounding volumes forming a natural tree structure. A collision query can then be performed using a traversal of the tree; initially checking if bounding volumes containing the object are overlapping, and if they are, performing a test on the objects themselves. The words *hierarchy* and *tree* are in the following sections analogous.

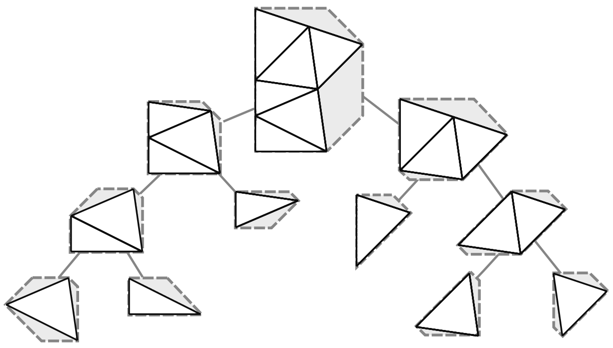


Figure 4.2: Bounding volume hierarchy with *kDOP bounding volumes* (Section 4.2.3), shown as gray dashed volumes around triangular primitives.

There are many ways to construct and traverse a bounding volume hierarchy. A good overview of desired BVH characteristics is given in Ericson [22], where a commonly used cost function, first introduced in Weghorst [47], is presented:

$$T = N_u \cdot C_u + N_p \cdot C_p + C_c, \tag{4.1}$$

where

T	is the total cost function for detecting intersections between two sets represented by bounding volume hierarchies,
N_u	is the number of overlap tests on bounding volumes that are performed,
C_u	is the cost of performing such an overlap test
N_p	is the number of primitives tested,
C_p	is the cost of performing such a primitive test,
C_c	is the cost of constructing a tree.

Both products in this governing cost function depends on the shape of the bounding volume. A loosely fitting bounding volume with a cheap overlap test might be faster than a tightly fitting bounding volume with an expensive overlap test, and vice versa. Additionally, the first product $N_u \cdot C_u$ depends on the balance of the tree, since a suboptimal division of the objects in the tree causes more subgroups to be traversed. However, reducing the cost of traversing the tree often means increasing the cost of constructing the tree, C_c.

A tree is here entirely described by the concept of *nodes*. A node denotes any location in a tree containing information. Nodes in the middle of a tree has references to both a *parent* node and one or more *children* nodes. A *root* node is a node without any parent, here a tree contains only one root node. A reference to a root node is thus a reference to a tree in its entirety. A *leaf* node, on the other hand, is a node without any children, making it the end of a branch in the tree. Since this tree contains no *loops*, this is formally known as a *rooted binary tree.* Here, to keep things simple, a leaf node always contains a single primitive belonging to an input mesh. It's worth noting that nodes can be paired irrespective of whether or not one of the nodes is a leaf node. A typically rooted binary tree structure is illustrated in Figure 4.3.

The bounding volume hierarchy is here presented in the order construction, traversal and bounding volume. A tree is initially built with assumptions on given input and intended usage. The assumptions made with the intent of doing collision detection on overlapping meshes is here attempted thoroughly exposed.

4.2.1 Bounding volume hierarchy construction algorithms

The different ways of constructing a bounding volume hierarchy can be divided into three categories:

- *Top-down* construction is the most popular. Put simply, the set of objects is first enclosed in an all-encompassing bounding volume. Then the objects inside this bounding volume are divided into groups according to a partitioning strategy successively until reaching the leaf nodes.
- *Bottom-up* construction starts with constructing bounding volumes for the leaf nodes containing the individual objects in the dataset. Then these bounding volumes are paired with each other successively depending on some pairing criteria, until the whole dataset is bounded.
- *Insertion* construction is a dynamic technique where according to some cost function associated with a branch or a tree, nodes are inserted into the existing tree structure.

In light of detecting intersections on overlapping meshes, a top-down approach like the one eventually shown in Algorithm 3 should suffice in comparison to the other construction strategies. There is no real need to pair individual cells or bounding volumes in the tree optimally as can be done with a bottom-up construction approach since it suffices that primitives are in a relative close proximity to each other. Likewise, inserting additional triangles into the tree efficiently is not something that is initially deemed necessary for the application at hand, making an insertion based technique superfluous.

Constructing a tree top-down can be done by first computing the bounding volumes (BVs) for each leaf node then *merging* these bounding volumes into a big volume containing all and then again split this volume into fitting partitions. Algorithm 3 constructs a tree with the input array *initial_nodes* as returned from Algorithm 2.

Top-down construction is possible using relatively few operations on the input data; especially when using bounding volumes that can be merged as described above. Merging bounding volumes like AABBs (Section 4.2.3) and kDOPs (Section 4.2.3) by finding the maximum bounding volume spanned by the coordinates of the input bounding volumes, produces what is theoretically the tightest AABBs and kDOPs encompassing the objects bounded by the original bounding volumes. This is not the case for volumes like the bounding spheres (Section 4.2.3) and various other bounding volumes, where for the tightest fit, the geometry of the objects have to be re-parsed for every node they appear in. However, if not settling for a tightest fit, a fairly good bounding volume can be obtained for spheres by just combining the radii and centers of the member nodes/bounding volumes.

As important as the bounding volumes themselves is the *partitioning strategy* used when the tree is constructed. The bounding volumes and primitives are in some way compared to each other before given a place in the tree structure. In a top-down approach when merging bounding volumes this is done by sorting the bounding volumes by some property like their midpoint or volume and splitting them into two sets of equal or different size, depending on some criteria. The partition strategy is determined by the functions *partition_nodes(...)* and *midpoint()* in Algorithm 3.

Since none of the cells in each mesh individually overlap and shape regularity (2.16) is assumed, sorting the primitives here on midpoints seems to be justified. In his pioneering work on kDOP bounding volumes, Klosowski [20] employs a similar strategy and shows that this gives best performance on the dataset tested. Also, assuming that the cells in the meshes are more or less uniformly spread throughout the domain, sorting the sets of cells with respect to the coordinate axis around which they are maximally spread initially, *an axis of maximum spread*, seems justified. Since this just involves comparing already available x, y, z-coordinates in the bounding volumes with a coordinate axis, the cost of this is relatively cheap. Cheap at least with respect to doing a more comprehensive comparison, like sorting around a calculated *free* axis of maximum spread in the data, which would involve calculating a dot product for every bounding volume when comparing and sorting these at all levels. With great variance in cell size, however, like for instance if a mesh is heterogeneously refined with respect to different features in a domain, this and more exotic partitioning strategies, like employing *surface area heuristics* (SAH) [48], should probably be considered.

Although not necessarily yielding the fastest tree to traverse, the tree given is fast to construct; thus hopefully striking a balance between traversal and construction time. Additionally, this strategy guarantees a *balanced structure* of the tree, eliminating some potential worst case behavior like that associated with an imbalanced tree structure, where a node can be at a depth n in a tree constructed for n objects. Figure 4.3 shows an actual tree constructed using the assumptions given above.

Algorithm 2 Computing bounding volumes for leaf nodes

```
for each $T \in \mathcal{T}$ :                                  ▷ $\mathcal{T}$ is the input mesh.
    $a_{\text{type}} \leftarrow$ Leaf                            ▷ $a$ denotes a leaf node object.
    $a_{\text{BV}} \leftarrow$ computeBoundingVolume($T$)
    $a_{\text{midpoint}} \leftarrow$ midpoint($T$)
    $a_{\text{n}} \leftarrow 1$                  ▷ $n$ is number of primitives contained within a.
    $a_{\text{OBJ}} \leftarrow T$            ▷ The primitive $T$ is made the object contained in $a$.
    leaf_nodes += $a$
return leaf_nodes
```

Algorithm 3 Tree construction algorithm

```
Require: $a$, leaf_nodes, $n$    ▷ $a$ is an input node; at first call the root node. $n$ is the
    size of leaf_nodes.
if $n \leq 1$ :                       ▷ If the BV contains one object it's a leaf node.
    a $\leftarrow$ leaf_nodes[0]
else
    $a_{\text{n}} \leftarrow n$                ▷ For the first call $n$ is the size of leaf_nodes.
    $a_{\text{type}} \leftarrow$ Node
    $a_{\text{BV}} \leftarrow$ mergeBoundingVolumes(leaf_nodes, n)
    $a_{\text{midpoint}} \leftarrow$ midpoint(leaf_nodes, n)
    seperation_axis $\leftarrow$ mostSeparatedAxisOfMidpoints(leaf_nodes, n)
    split_index $\leftarrow$ partitionNodes(leaf_nodes, n, separation_axis)
    constructTree($a_{\text{left}}$, leaf_nodes[0, ..., split_index], split_index)
    constructTree($a_{\text{right}}$, leaf_nodes[split_index, ... ,n], n - split_index)
```

mergeNodes(...) constructs a BV by merging the nodes in the input array.
mostSeparatedAxisOfMidpoints(...) returns an axis where the BVs are most spread.
partitionNodes(...) sorts the input array *leaf_nodes* around the found axis.
constructTree(...) calls Algorithm 3 again, with a child a_{right} or a_{left} of the initial node, and a partition of *leaf_nodes* consisting of elements up to the *split_index*, or from the split index to the end of the array as input. The size n of *leaf_nodes* becomes the size of this partition for each consecutive call.

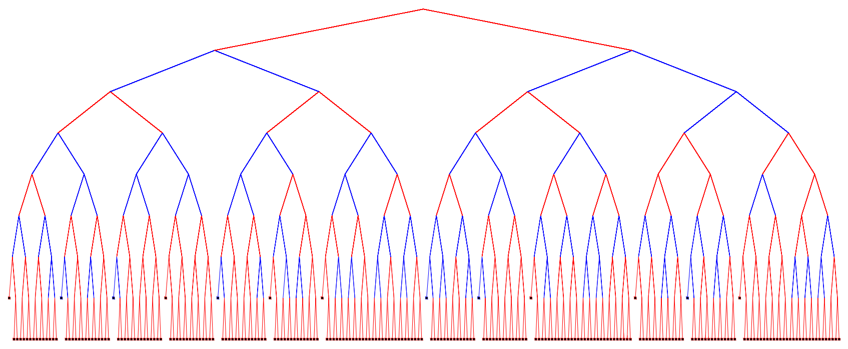


Figure 4.3: Actual constructed tree for the sample background mesh $\mathcal{T}_1$ found in Chapter 3. Colors indicate around which *axis of maximum spread* the bounding boxes have been sorted, here coordinate axes: red denotes the x-axis and blue the y-axis. Generation code for this three is found in A.5.

4.2.2 Traversing the bounding volume hierarchy

Once having constructed a tree for each mesh, the trees are queried for information. Depending on the specific query – be it finding a closest point to a mesh, checking for a first intersection by another mesh, or detecting all collisions – a fitting traversal routine of the tree structure has to be made. Being mainly interested in detecting all the intersected cells in the background mesh $\mathcal{T}_0$ and also obtaining information about which cells in $\mathcal{T}_0$ intersects which cells in $\mathcal{T}_2$, a *simultaneous traversal* of the trees belonging to each mesh is made. Starting with the root node in each tree (denoted by a and b, in the code below), the trees are descended into in a simultaneous fashion, meaning that only regions with overlapping bounding volumes are evaluated for collisions. A *stack* data structure is in Algorithm 4 used to make the traversal non-recursive. The *descendRule(...)* function is used to determine which child node to descend into from a parent node. Here a descent is made into the bounding volume with the biggest volume.

Algorithm 4 Tree traversal algorithm

```
Require: $a, b$                                          ▷ The root nodes in each tree.
  stack_a = 0
  stack_b = 0
  while True :
      if $a_{\mathrm{BV}} \cap b_{\mathrm{BV}} \neq \emptyset$ :                  ▷ Bounding volumes intersect.
          if is_leaf($a$) and is_leaf($b$) :
              if $a_{\mathrm{OBJ}} \cap b_{\mathrm{OBJ}} \neq \emptyset$ :                  ▷ Primitives intersect.
                  collsion_set ← pair($a_{\mathrm{OBJ}}, b_{\mathrm{OBJ}}$)
          else
              if descendRule($a, b$) = $a$ :
                  stack_a ← $a_{\mathrm{right}}$
                  stack_b ← $b$
                  $a$ ← $a_{\mathrm{left}}$
              else
                  stack_a ← $a$
                  stack_b ← $b_{\mathrm{right}}$
                  $b$ ← $b_{\mathrm{left}}$
      if stack_a = $\emptyset$ and stack_b = $\emptyset$ :
          break
      else
          $a$ ← top(stack_a)
          $b$ ← top(stack_b)
   return collision_set
```

collision_set is here a data structure keeping track of collisions between pairs of objects, like for instance a map or vector of pairs.

Note that in Algorithm 4, very little geometrical information about the bounding volumes and primitives is required. These traits effectively illustrate the degree of "geometric obliviousness" a BVH admits; only when constructing the bounding volumes and testing the primitives themselves is specific information about an objects topology needed.

More advanced methods to traverse a bounding volume hierarchy exist: one can for instance traverse a hierarchy by *front tracking* [22], that is, starting at a node in the middle of the hierarchy instead of at the root node based on some statistical or historical heuristic. Since detecting all intersections is the main goal here, a simultaneous traversal as performed above, however, seems sufficient. Doing a simultaneous traversal ensures some *locality* during the query to the hierarchy, meaning that if two bounding volumes containing many primitives are colliding, the traversal is made so that all the intersections contained within these bounding volumes are found before returning to a node higher up in the hierarchy, like the root node.

4.2.3 Bounding volumes

Of the many bounding volumes available, three classes of bounding volumes are investigated here: bounding spheres, axis aligned bounding boxes (AABB) and k oriented polytopes (kDOP). These bounding volumes can all be merged in a satisfactory manner and can therefore be incorporated into a fast construction algorithm as the one shown in Algorithm 3. Other possible bounding volumes are, to name a just a few, are oriented bounding boxes [44], swept spheres [49], cones and cylinders. Each bounding volume has associated geometrical information and two essential functions; a constructor and an overlap test, all with varying storage requirements and computational costs.

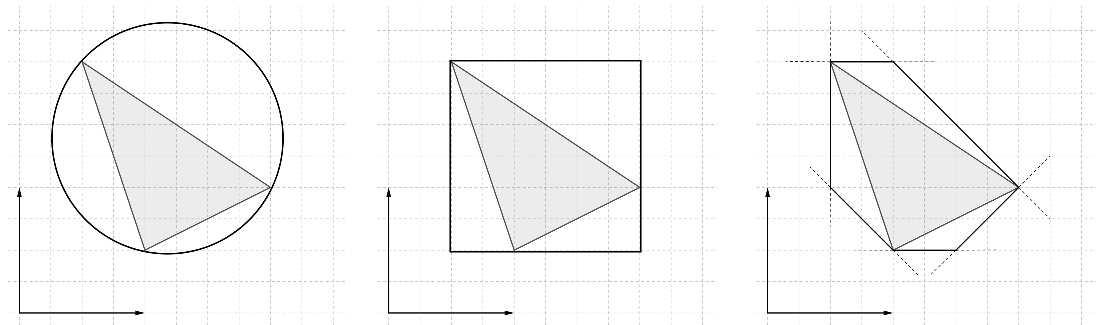

(a) Bounding Volume Sphere (BVSphere **(b)** Axis Aligned Bounding Box (AABB) **(c)** 8th Discrete Oriented Polytope (KDOP)

Figure 4.4: The bounding volumes considered. Stipled lines in the background denote coordinate axis x and y.

Bounding spheres

A bounding sphere is the simplest geometrical bounding volume. To store a bounding sphere, only a radius and a point denoting a center is needed, making it one of the cheapest volumes to store in memory. If two bounding spheres overlap the distance between their centers are less then the sum of their radii combined. Although simple, the overlap test given in Algorithm 5 is not necessarily cheaper than the others considered, it does however require fetching the least data. As for the construction of such a sphere,

Algorithm 5 Overlap test for bounding spheres

```
Require: $a, b$                                    ▷ Bounding sphere volumes.
  $d \leftarrow \text{center}(a) - \text{center}(b)$
  $r_s \leftarrow \text{radius}(a) + \text{radius}(b)$
  return $d \cdot d \leq r_s^2$          ▷ Squared distance less than squared sum of radii.
```

the Ritter sphere [50] algorithm provides a cheap construction of a sphere encompassing an arbitrary number of points. Because of computational efficiency, this algorithm does not produce the tightest bounding sphere possible on the data; however, Ritter [50] shows that the computed radius is only about 5% bigger than the ideal radius of the circumsphere containing the given points.

Axis aligned bounding boxes

The most popular choice of bounding volume, the *axis-aligned bounding box*, has several sought features. A straight forward overlap test and simple construction makes for easy implementation, and its performance for many applications is sufficient. Constructing an AABB is done by gathering the minimum and maximum of the objects vertices on each coordinate axis; in 3D the Cartesian product of the intervals $[min_i, max_i]$ for $i = x, y, z$. If two volumes do not overlap, then either the minimum in one volume is less than the maximum, or the maximum in one is less than the minimum in the other.

Algorithm 6 Overlap test for AABB bounding volumes

```
Require: $a, b$                                    ▷ AABB bounding volumes.
  for each axis $\in \{(1,0,0), (0,1,0), (0,0,1)\}$ :
      if $a_{axis,\max} < b_{axis,\min}$ or $a_{axis,\min} > b_{axis,\max}$ :
          return False
  return True
```

k discrete-oriented polytopes

With bounding spheres as an exception, most bounding volumes are convex polyhedrons. Polyhedral bounding volumes can be represented as intersections of sets of half-spaces - AABBs are the intersection of six half-spaces. If the bounded object is polyhedral, the tightest possible convex bounding volume is the *convex hull* of the object. The data needed to store such a structure is a normal and a distance (from the origin) for each half-space in the set, which for a complex object tend to be many.

To borrow some of the properties of the convex hull and to retain some of the properties that are computationally easier to deal with, the *discrete-oriented polytope* introduced in Klosowski [20], limits the normal components needed store and compute a convex hull to a predefined set, typically $\{-1, 0, 1\}$. A set of k axes made from these components is then shared among all the kDOP bounding volumes.

To compute the kDOP of a given object, a pass over every vertex in the object is made and for every vertex the dot product of a the vertex with the kth unit axis is computed. Storing the maximum and minimum value of the this dot product for every axis effectively defines the kth plane in the bounding volume. This is illustrated in Algorithm 7.

Algorithm 7 Construction algorithm for a kDOP bounding volume

```
Require: polytope
  k = 0                                                    ▷ kDOP volume.
  for each $v_i \in$ polytope :
      for each axis $\in K_{\text{axes}}$ :                        ▷ $K_{\text{axes}}$ is the set of axes.
          $n_{\text{val}} \leftarrow v_i \cdot axis$
          $k_{\text{axis,max}} \leftarrow \max(n_{\text{val}}, k_{\text{axis,min}})$
          $k_{\text{axis,min}} \leftarrow \min(n_{\text{val}}, k_{\text{axis,min}})$
  return $k$
```

Here v_i are vertices in the input polytope.

The overlap test of kDOP examines whether the minimal distance with regards to its axis of the kth plane in the first volume is larger than the maximum distance of the kth plane in the second volume. Just as in the case of the AABB, if this is true for any of the kth planes, then bounding volumes are overlapping. Testing the overlap of kDOP bounding volumes is thus done as in Algorithm 6, only with different axes.

An example of this bounding volume is the 6-DOP, with normals in both directions along the axes, $(\pm1, 0, 0)$, $(0, \pm1, 0)$, and $(0, 0, \pm1)$. The 6-DOP is an ordinary AABB. The 8-DOP, shown in Figure 4.4c in two dimensions, has the axes $(\pm1, \pm1, \pm1)$. Note that kDOPs are named after the number of defining axes in the 3D case. Also note that some kDOP can also *embed* other kDOPs, since for instance the axes of a 6DOP, $(\pm1, 0, 0)$, $(0, \pm1, 0)$, can also be among the 26 axes defining a 26DOP.

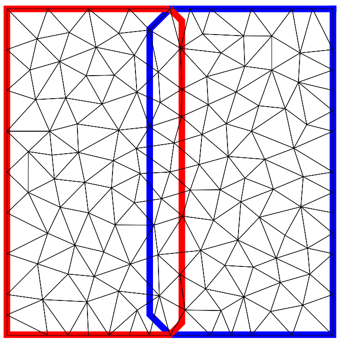
The data defining the planes in Algorithm 7 does not coincide with the geometrical interpretation of them. This is because it is more computationally efficient to work with normals where the axes are of unit length than a normal of, say, unit length. The point

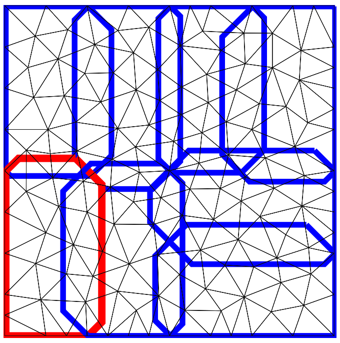
$\mathbf{g}_{axis}$ geometrically defining the plane for $axis = (N_x, N_y, N_z)$ is obtained by the formula

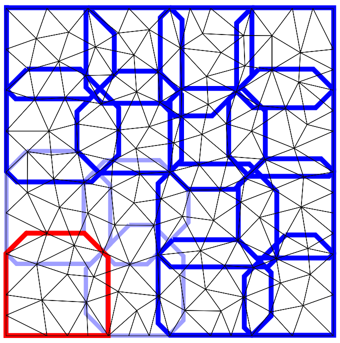
$$\mathbf{g}_{axis} = k_{axis} \frac{1}{|N_x| + |N_y| + |N_z|} (N_x, N_y, N_z),$$

which is useful for instance when drawing these bounding volumes. When testing bounding volumes for overlap using the coefficient k_{axis} this is of no consequence, since the planes tested against each other are equally scaled.

When using kDOP bounding volumes to enclose polytopes like triangles and tetrahedrons, the bounding volume has the ability to fit these primitives perfectly. The bounding volume coincides with the primitive if the sides of a triangle or a tetrahedron are aligned to the same axes as that of the kDOP and k is large enough. This is the case for green bounding volume in Figure 4.5d.


(a) Bounding volumes containing more than 100 cells.


(b) Bounding volumes containing around 40 cells.


(c) Bounding volumes containing around 15 cells.

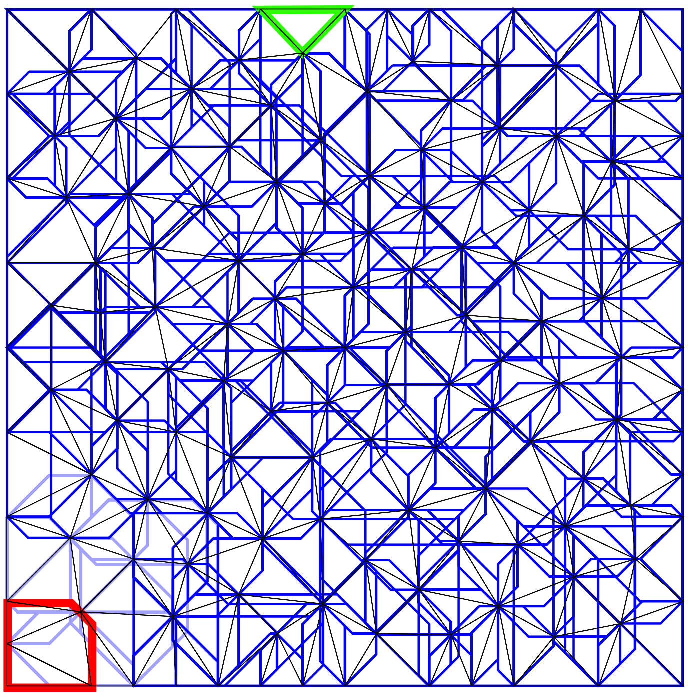

(d) Bounding volumes containing around 2 cells. The green BV contains one and fits its primitive perfectly.

Figure 4.5: Different levels of hierarchy with 26DOP 2D bounding volumes on $\mathcal{T}_1$. Some blue volumes have been made translucent around red volumes. Code for generating this illustration is given in A.6. Note that the planes shown here does not actually coincide with the planes that are stored in Algorithm 7.

4.2.4 Detecting the intersection of primitives

Traversing a bounding volume hierarchy usually ends in testing if two objects locally collide. In the traversal Algorithm 4 above this is denoted as the intersection $a_{\text{BV}} \bigcap b_{\text{BV}}$. The computational cost of this procedure, C_p in the cost function (4.1), depends on many factors, namely the type of geometry tested, the quality of the algorithm and its geometrical accuracy. For a tetrahedron-tetrahedron collision test, a popular algorithm is Ganovelli [19]. This algorithm makes use of *Separating Axis Theorem*, stating that two separated closed convex sets are always separated by a hyperplane. For triangle-triangle intersection, a popular algorithm is given in Möller [51].

Properties of the geometrical machinery should not be a source of error in the numerical simulation, and can be made so that it *never* will be given proper input. A bounding volume hierarchy can be constructed so that it never fails to give sufficient collision candidates based on the input data. With for instance kDOP bounding volumes, this can be done by just adding an appropriate quantity to the coefficients k_{axis} scaling the normals defining the bounding volume. These candiates are only one of many concerns in the big picture though, and many It is not uncommon that a code of a particular algorithm for primitive-primitive test is ambiguous, that is, having a tendency to produce *collide(a,b)* $\neq$ *collide(b,a)*, for two primitives a and b. Sometimes this is surely because of errors in an implementation, but it is also because of the fact that many widely used algorithms use different data associated with each primitive to pose geometric predicates. Depending on the order they are put into a routine, these algorithms react to numerical inaccuracies and geometrical degeneracies differently and this makes many of these hard to debug and explain. Many algorithms, perhaps especially those described as *fast*, are intended for use in real-time contexts, where these tests are performed more as an indication of collision than as an exact geometric description, and are typically performed so many times consecutively that the behavior of the game or design tool is consistent.

The Computational Geometry Algorithms Library, CGAL [18] is an extensive library of algorithms exemplifying these inherent inaccuracy concerns when making geometrical predicates. Exemplifying this particularly through offering different geometry kernels with different geometric accuracy as back-ends to many of their algorithms. Cutting the investigation of numerical robustness short, many of the primitive-primitive tests relevant for detecting intersections on overlapping meshes are in the implementation made available with a selection of tests (4.4.1) - some of them from CGAL and some of them *fast*. Going into more specific numerical robustness issues is considered out of the scope of this thesis. Lutz Kettner et al. *Classroom examples of robustness problems in geometric computations* [52] shows several illuminating examples of robustness problems in geometric computations.

4.3 An advancing front algorithm

Advancing front algorithms are commonly used when constructing meshes. Meshing a domain is then done by progressively adding cells starting for example at the boundaries. The *front* is in this regard the border between the meshed and unmeshed region, and this front is iteratively *advanced* to cover the whole mesh. A Delanuay triangulation, mentioned in Section 2.2.3, is for instance often constructed by an advancing front point-placement strategy [28]. The concept of an advancing front is here employed to detecting the intersection between two meshes, and the algorithm presented is that of Martin J. Gander [21]. Quantities derived when using Gander's algorithm can further be used to generate a triangulation of the intersection of the involved meshes, as is done by a similar algorithm in Farrell [5]. Computing this union is not initially relevant for the application at hand and is therefore not presented.

Section 4.1.2 explained the basic idea of a front advancing algorithm for detecting intersections; mesh connectivity information is used to move from an intersection of two triangles, here called R_T and B_T, to intersections of neighboring triangles. Let these triangles belong to the respective triangulations $R_{\mathcal{T}}$ and $B_{\mathcal{T}}$. In Gander's [21] algorithm, data denoting on which facet a triangle is intersected is used to efficiently move on to successive candidates. If a triangle B_T is intersected by R_T on facet $F_{B,i}$, then the neighbor of B_T corresponding to $F_{B,i}$, N_T, is also checked for intersections with R_T. Continuing this way also for the neighbors of R_T - that is, checking them for intersections with B_T and its neighboring N_T's - eventually gives us complete information about the overlap between the two meshes $B_{\mathcal{T}}$ and $R_{\mathcal{T}}$.

Since only information about neighboring triangles is used to traverse the domain, an initial input of two triangles already known to intersect is required for the algorithm to successfully run. Finding these triangles is typically done by a brute force pass on the data, i.e. picking an arbitrary triangle in one mesh and checking it for a first intersection by going through the elements of the other. In a typical scenario this pass on the data is of negligible computational cost, but it is however easy to envision cases where a brute force approach would become costly or fail to supply sufficient amounts of data. This can happen for instance if the meshes aren't completely overlapping. Then a large number of cells in each mesh might have to be checked for intersections; leaving in a potential constructible worst case of order $\mathcal{O}(n^2)$ in the algorithm. This can be ameliorated by using some heuristic to find an appropriate *seed* cell or by using a partitioning method like that of a BVH to be guaranteed a candidate at a fair cost. Similar complications will naturally arise if one or both of the domains is not simply connected.

Algorithm 8 is made for intersections on meshes containing triangular cells. Farrell [5] presents a similar algorithm for meshes containing tetrahedral cells. Generalizing this algorithm for efficiently detecting intersections between meshes of different topological dimension, like a tetrahedral and a triangular domain, seems to be a non-trivial task.

Similar numerical concerns as those expressed in Section 4.2.4 also apply here. However, here such inaccuracies might be detrimental to the whole process. When quantities derived about the intersection of cells are used to queue triangles like described above, triangles needed for the algorithm to advance might be left out completely, and halt the process before it has parsed the whole domain. This property can arguably be positive; since when numerical inaccuracies produce results far from expected outcome this necessitates resolving the numerical inaccuracies first-hand. Doing so limits their potential propagation into subsequent operations.

The process of successively checking neighbors of intersecting cells is here realized with the use of both global and local queues and sets of indexes denoting which cells have already been processed. These queues and sets (flags) are denoted $\square_{\text{Blue queue}}$, $\square_{\text{Red queue}}$, $\diamondsuit_{\text{Local red queue}}$, $\diamondsuit_{\text{Blue flags}}$, $\diamondsuit_{\text{Red flags}}$, $\angle_{\text{Blue facet flags}}$ in Algorithm 8, and Some of the geometric quantities involved are shown in Figure 4.6.

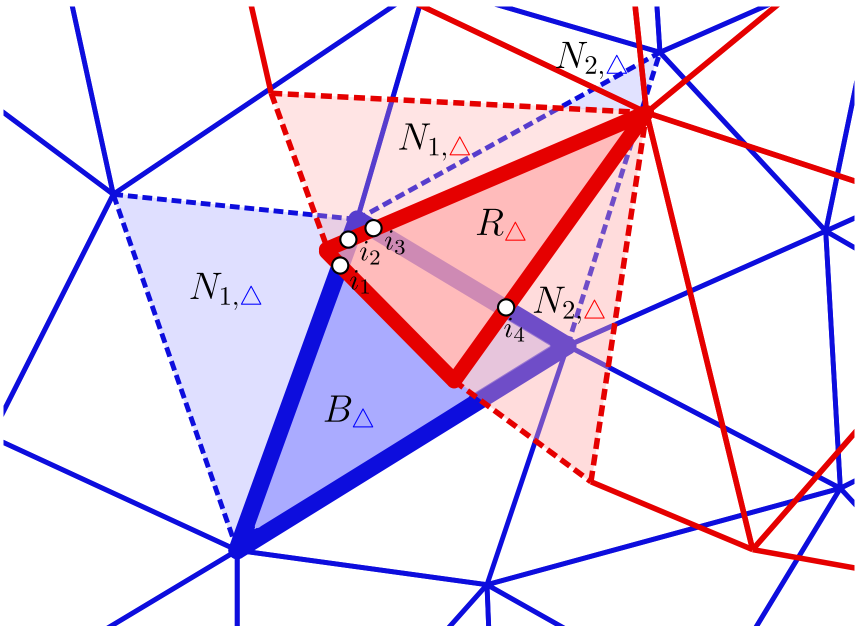


Figure 4.6: Entities involved when checking $B_{\triangle}$ for intersections. Since the facet of $B_{\triangle}$ corresponding to neighbor $N_{1,\triangle}$, is intersected by $R_{\triangle}$ at i_1 and i_2 both $B_{\triangle}$ is added to the blue queue and $R_{\triangle}$ is again added to the red queue. Likewise for the other neighbor $N_{2,\triangle}$ of $B_{\triangle}$. The triangles $N_{1,\triangle}$, $N_{2,\triangle}$ are checked for intersection with $B_{\triangle}$ and, if intersecting, added to the set of intersections. Depending on where these triangles intersect the facets of $B_{\triangle}$, these are added to the red queue.

Algorithm 8 Advancing front algorithm

```
Require: $B_{\mathcal{T}}, R_{\mathcal{T}}, B_{\text{seed}}, R_{\text{seed}}$
  $B_{\triangle} \leftarrow B_{\text{seed}}$
  $R_{\triangle} \leftarrow R_{\text{seed}}$
  if $\neg\text{Intersect}(B_{\triangle}, R_{\triangle})$ :
      $B_{\triangle}, R_{\triangle}$ = Brute-force first intersection
  $\text{push}(B_{\triangle}, \square_{\text{Blue queue}})$
  $\diamond_{\text{Blue flags}} \leftarrow \{\}$
  $\text{push}(R_{\triangle}, \square_{\text{Red queue}})$
  while $\square_{\text{Blue queue}} \neq \emptyset$ :
      $\diamond_{\text{Red flags}} \leftarrow \{\}$
      $\angle_{\text{Blue facet flags}} \leftarrow \{\}$
      $B_{\triangle} \leftarrow \text{pop}(\square_{\text{Blue queue}})$
      $R_{\triangle} \leftarrow \text{pop}(\square_{\text{Red queue}})$
      $\text{push}(R_{\triangle}, \diamond_{\text{Local red queue}})$
      while $\diamond_{\text{Local red queue}} \neq \emptyset$ :
          $L_{\triangle} \leftarrow \text{pop}(\square_{\text{Local red queue}})$
          if $\text{Intersect}(B_{\triangle}, L_{\triangle})$ :
              $\mathcal{I} \mathrel{+}= \text{pair}(B_{\triangle}, L_{\triangle})$
              for each $N_{\triangle} \in \text{neighbors}(L_{\triangle})$ :
                  if $N_{\triangle} \notin \diamond_{\text{Red flags}}$ :
                      $\text{push}(N_{\triangle}, \diamond_{\text{Local red queue}})$
                      $\diamond_{\text{Red flags}} \mathrel{+}= N_{\triangle}$
                  if $\text{IntersectsFacet}(B_{\triangle}, N_{\triangle})$ :
                      $\angle_{\text{Blue facet flags}}(N_{\triangle}) \mathrel{+}= L_{\triangle}$
      for each $N_{\triangle} \in \text{neighbors}(B_{\triangle})$ :
          if $N_{\triangle} \notin \diamond_{\text{Blue flags}}$ and $\exists R_{\triangle} \in \angle_{\text{Blue facet flags}}(N_{\triangle})$ :
              $\text{push}(N_{\triangle}, \square_{\text{Blue queue}})$
              $\text{push}(R_{\triangle}, \square_{\text{Red queue}})$
              $\diamond_{\text{Blue flags}} \mathrel{+}= N_{\triangle}$
   return $\mathcal{I}$
```

Explanation of symbols by order of appearance:

$B_{\mathcal{T}}$, $R_{\mathcal{T}}$	are triangulations of a domains Ω_1, "blue mesh", and Ω_2, "red mesh",
$B_{\triangle}$, $R_{\triangle}$	is a triangle in $B_{\mathcal{T}}$ and $R_{\mathcal{T}}$,
$B_{\text{seed}}, R_{\text{seed}}$	are seeds for these,
Intersect($\triangle$, $\triangle$)	returns true if two triangles intersect,
push($\triangle$,queue)	is an operator enqueuing a triangle in a queue,
$\square_{\text{Blue queue}}$, $\square_{\text{Red queue}}$	is a *first-in first-out* queue of triangles from $B_{\mathcal{T}}$ and $R_{\mathcal{T}}$,
$\diamond_{\text{Blue flags}}$, $\diamond_{\text{Red flags}}$	is sets of *flags* denoting whether or not a triangle in $B_{\mathcal{T}}$, $R_{\mathcal{T}}$, has been parsed,
pop(queue)	is an operator taking an element at the front out of the queue,
$\diamondsuit_{\text{Local red queue}}$	is a first-in first-out queue of triangles from $R_{\mathcal{T}}$,
$L_{\triangle}$	is a triangle local to the while loop the attains both initial $R_{\triangle}$ and $N_{\triangle}$.
$\mathcal{I}$ += pair$(B_{\triangle}, L_{\triangle})$	adds a pair of triangles intersecting to the set of intersections
neighbors($\triangle$)	returns ordered neighbors of the input triangle, clockwise or anti-clockwise. This order is used when associating neighbor triangles with facet-intersections from previous iterations,
$\angle_{\text{Blue facet flags}}$	is a set of flags denoting whether or not a facet in a triangle $B_{\triangle}$ is affected by collision with a triangle $L_{\triangle}$,
IntersectsFacet$(B_{\triangle}, N_{\triangle})$	returns true if the facet in $B_{\triangle}$ *corresponding* to neighbor $N_{\triangle}$ of a triangle $L_{\triangle}$ is intersected by $L_{\triangle}$,
$\angle_{\text{Blue facet flags}}(N_{\triangle})$ += $L_{\triangle}$	flags the facet corresponding to $L_{\triangle}$ in $B_{\triangle}$ with information about $L_{\triangle}$, these are treated as flags since they are denoted by the ordering of the neighbors of the triangle in the original implementation. By this flagging procedure; in Figure 4.6 the triangle $N_{1,\triangle}$ would be associated with $N_{1,\triangle}$.

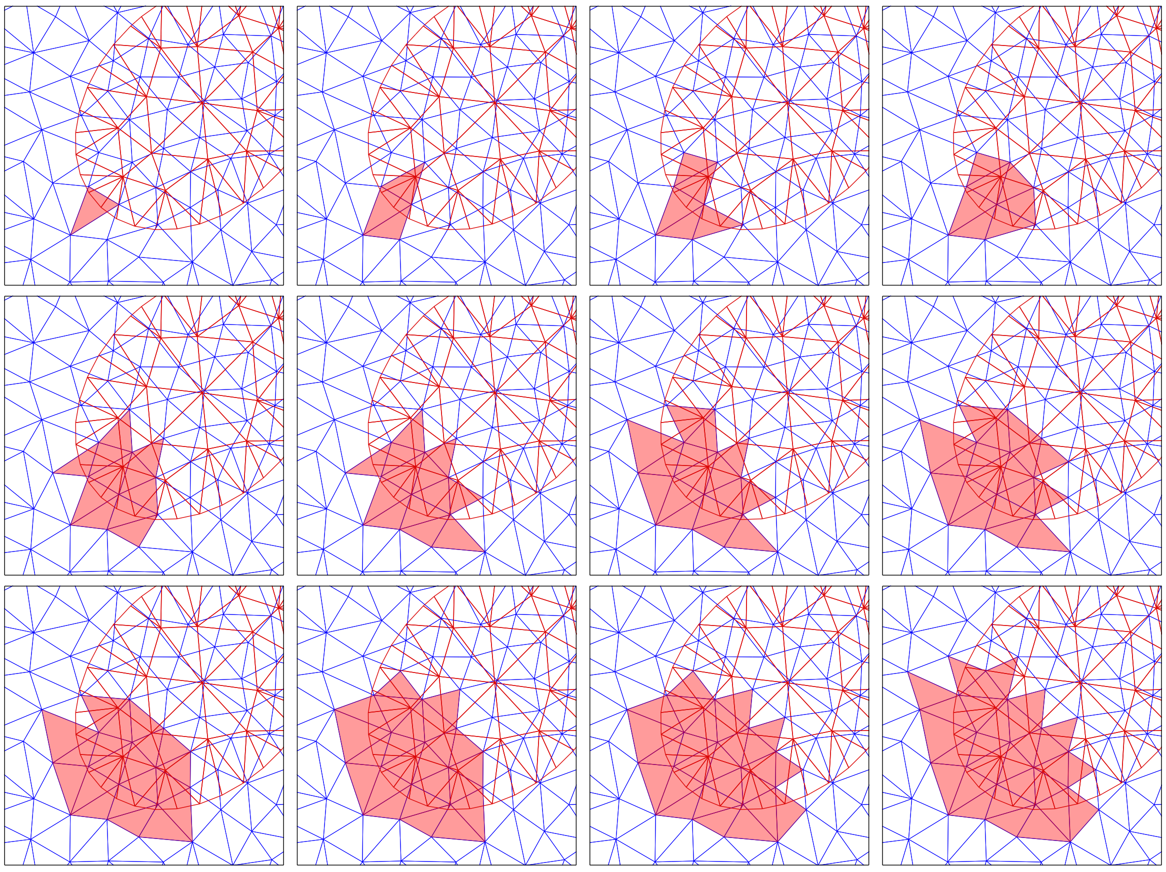


Figure 4.7: Snapshot at every 10th detected intersection on sample meshes by the advancing front algorithm, Algorithm 8.

4.4 Implementation of algorithms

All the major implementations in this thesis are written in C++. Minor code for generating example meshes and some tests have been written in Python. The codes have all been written towards the FEniCS project PSE (Problem Solving Environment) [53], and more specifically towards the module Dolfin [54] and library LibCutFEM [14], both operating within this environment. The computational geometry algorithms are here presented without an easily installable stable package. Due to the active development of several of the frameworks wherein these algorithms are supposed to work, compiling such a stable package would be impossible within the time frame of this development. The C++ source and header files of core algorithms, along with rudimentary comments in the code, are available at `https://github.com/tomana/compgeom_olm/`.

Apart from using the FEniCS framework, the C++ open source toolkit openFrameworks [55] has been used at top level of applications during the development of these algorithms. This toolkit has enabled the integration of all the relevant functionality of Dolfin [54] and LibCutFEM [14] within a consistent execution and rendering context based on OpenGL [56], and has been useful for both visual debugging and the generation of figures directly from numerical results.

4.4.1 Implementation of a bounding volume hierarchy

C++ code implementing algorithms mentioned in the following is available at `https://github.com/tomana/compgeom_olm/tree/master/bvh`

The implementation of a bounding volume hierarchy is here built around the concept of easily being able to swap out the type of bounding volumes used. To this end, many STD[57] *templates* are employed throughout the code. These template classify both different geometrical entities, triangles or tetrahedrons, and bounding volumes. The use of templates together with *function overloading* allows for fair comparison between the bounding volumes. Additionally, employing templates like this avoids testing bounding volume or primitive types at run time, since the code is *unrolled* in the compiler. No major optimization of data structures or parallelization techniques have been attempted.

Implementing efficient tree construction

To swap types of bounding volume efficiently, a template `class BVolume` is used throughout the implementation and the function for tree construction accepts an array of bounding volumes already constructed for each primitive.

The inner workings of the *partition_nodes(...)* function appearing in Algorithm 3, relies on the Standard Library [57] `nth_element_sort` algorithm. This algorithm minimally sorts the input array with a given coordinate axis as pivot. Elements are arranged in two equally sized sets about the pivot axis; all elements in one set having larger values with respect to the given axis than the elements in the other. The order of the elements *within* the two separate sets is, however, arbitrary.

Implementing a dynamic traversal routine

In the implementation of the traversal routine Algorithm 4, the template for the bounding volume `class BVolume` appears just as in the construction algorithm. This implementation also makes use of the template `class Container`, which denote what kind of data structure is used for the geometric data; be it a mesh, array of cells or a point cloud. Functionality for the last two containers is not yet implemented.

Once an intersection of two primitives has been detected between the two meshes, it is stored in an `std::unordered_map` containing *key-value* pairs, where unique keys correspond to an index denoting a cell in the background mesh and the values one or several indexes of intersecting cells in the overlapping mesh.

Availability of bounding volumes and intersection tests in the implementation

In the implementation a selection of bounding volumes and primitive intersection tests are available in the constructor call:

```
tree = new BoundingVolumeTree(*mesh, "PrimitiveTest", "BoundingVolume", true);
```

Calling this constructor with third argument `"BSphere"` constructs a bounding sphere based on Ritter's [50] algorithm. Arguments `"AABB"`, `"DOPx"` for `x = 6, 8, 12, 14, 18, 26`, calls implementations of bounding volumes as described above in Algorithm 6 and Algorithm 7 in Section 4.2.3.

The second argument denotes what kind of technique for primitive intersection test to use. For tetrahedron-tetrahedron and triangle-triangle intersection tests, the available are

- `"Dolfin"`, an implementation of the algorithm given in Ganovelli [19], described in part in Section 4.2.4. This algorithm does not work if the tetrahedron is degenerate.
- `"SimpleCartesian"` is an implementation based on two CGAL[18] functions checking if a point is inside a tetrahedron and whether or not a face of one triangle intersects the other. This method is slightly slower than the above, but works on degenerate tetrahedra.
- `"ExactPredicates"` is the same CGAL implementation using the `ExactPredicatesInexactConstructions` geometry kernel of CGAL. When it comes to geometric accuracy, this algorithm is exact on the given input [18]. This is slower than the `"SimpleCartesian"` test by orders of magnitude.
- `"None"` is not using a primitive test at all, returning a positive intersection for all primitives with overlapping bounding volumes.

The last boolean argument in the constructor turns on the enclosing of leaf nodes in a an additional `26DOP` bounding volume.

4.4.2 Implementation of an advancing front algorithm

A modified implementation of Gander's algorithm [21] can be found here:
`https://github.com/tomana/compgeom_olm/tree/master/frontadvancing`

This code is based on a port of the original implementation by Martin J. Gander in Matlab to C++ by the MeshKit[58] and Mesh Oriented dAtaBase (MOAB)[59] teams.

The code has here been further modified to natively support Dolfin[54] data structures, and it is also modified to run in linear time ($\mathcal{O}(n)$). Descriptions of these specific changes and other small modifications are available as comments in the files.

For clarity, *intersect(...)*, *neighbors(...)* and *neighbor_intersecting_facet(...)* are as they appear in Algorithm 8 split into three separate functions. In the implementation these are derived from data returned from one function.

As with the bounding volume hierarchy, no major optimization of data structures in this algorithm has been attempted, and some redundant functionality is left within the implementation, which might limit the performance of the code.

4.5 Benchmarking implementations

All these tests have been performed on an system with an Intel Core i7 processor Q 820 1.73GHz and 6 GB 1333MHz DDR3 RAM running in a single thread.

4.5.1 Benchmarking different bounding volumes in 3D

Opting for clarity, the benchmarks presented here is limited to three geometrical cases. The first being a “real world” case: a spiral submitted into a domain of a sphere merged with a cylinder, mimicking an aneurysm geometry with an inserted stent. The second and third being simple structured meshes with varying degrees of shape regularity.

4.5.2 Benchmark on aneurysm example

The domain (Figure 4.8) was constructed in two parts; the spirals (blue) were made by sweeping circles along spiral splines in *Pyformex*[35] and the cylinder combined with a sphere (gray) was made using *Constructive Solid Geometry* (CSG) [60], both meshed in Netgen [32].

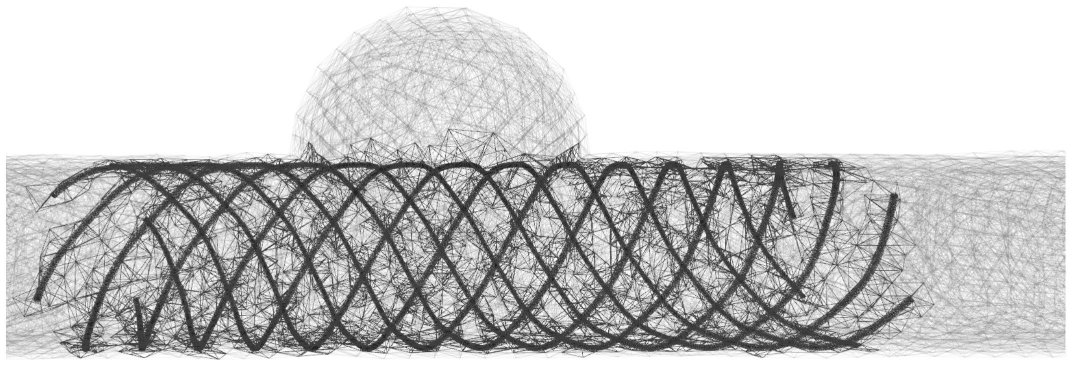


Figure 4.8: Intersections computed in unstructured meshes. Cells in background mesh (light gray): 80602, colliding mesh (black): 125033, intersection (mid-tone gray): 3397

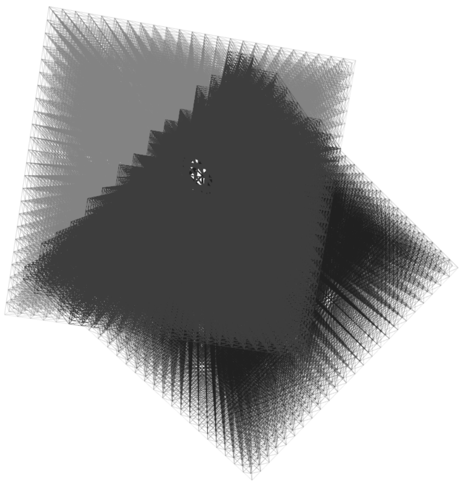
Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$1.52 \cdot 10^7$	$2.27 \cdot 10^6$	$2.45 \cdot 10^5$	$2.02 \cdot 10^6$	420ms	1970ms	2390ms
AABB	$3.89 \cdot 10^6$	$1.08 \cdot 10^6$	$2.45 \cdot 10^5$	$8.36 \cdot 10^5$	420ms	883ms	1303ms
6DOP	$4.29 \cdot 10^6$	$1.08 \cdot 10^6$	$2.45 \cdot 10^5$	$8.36 \cdot 10^5$	391ms	866ms	1257ms
8DOP	$6.89 \cdot 10^6$	$8.54 \cdot 10^5$	$2.45 \cdot 10^5$	$6.09 \cdot 10^5$	431ms	782ms	1213ms
12DOP	$3.02 \cdot 10^6$	$6.49 \cdot 10^5$	$2.45 \cdot 10^5$	$4.04 \cdot 10^5$	459ms	584ms	1043ms
14DOP	$2.91 \cdot 10^6$	$6.13 \cdot 10^5$	$2.45 \cdot 10^5$	$3.68 \cdot 10^5$	479ms	569ms	1048ms
18DOP	$2.87 \cdot 10^6$	$6.18 \cdot 10^5$	$2.45 \cdot 10^5$	$3.74 \cdot 10^5$	521ms	561ms	1082ms
26DOP	$2.42 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	606ms	494ms	1100ms

Table 4.1: Benchmark of different bounding volumes on aneurysm example

As seen in Table 4.1, there is not a big difference in performance between AABBs and kDOP bounding volumes in terms of total time on this dataset. The relationship construction versus traversal time is as expected, where higher construction time (**Constr.**) generally means lower traversal time (**Trav.**), except when it comes to the less tight bounding spheres. The traversal time is consistent with the lower number of primitive queries (**P q's**). The number of primitive queries, however, is not in a direct relationship with the number of traversals (**Trav's.**), where for instance the number of traversals are higher in the case of a 14DOP than in the case of a 18DOP, yet the number of primitive queries for a 18DOP is greater than the former.

4.5.3 Benchmark on structured mesh, regular split cubes

A structured mesh consisting of tetrahedrons that in pairs make up a cube (Figure 4.9), is a geometry often used in numerical simulation.


(a) Overview

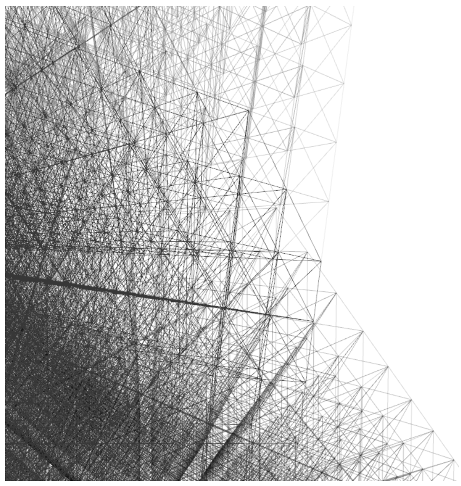

(b) Detail

Figure 4.9: Intersections computed on regular meshes. Cells in background mesh (black): 82944, colliding mesh (light gray): 82944, intersection (mid-tone gray): 52632

Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$5.56 \cdot 10^7$	$8.17 \cdot 10^6$	$9.39 \cdot 10^5$	$7.23 \cdot 10^6$	329ms	8500ms	8829ms
AABB	$1.48 \cdot 10^7$	$3.76 \cdot 10^6$	$9.39 \cdot 10^5$	$2.83 \cdot 10^6$	333ms	3782ms	4115ms
6DOP	$1.61 \cdot 10^7$	$3.76 \cdot 10^6$	$9.39 \cdot 10^5$	$2.83 \cdot 10^6$	337ms	4175ms	4512ms
8DOP	$2.11 \cdot 10^7$	$2.47 \cdot 10^6$	$9.39 \cdot 10^5$	$1.53 \cdot 10^6$	381ms	3351ms	3732ms
12DOP	$9.8 \cdot 10^6$	$1.65 \cdot 10^6$	$9.39 \cdot 10^5$	$7.14 \cdot 10^5$	445ms	2057ms	2502ms
14DOP	$1.06 \cdot 10^7$	$1.78 \cdot 10^6$	$9.39 \cdot 10^5$	$8.41 \cdot 10^5$	419ms	2216ms	2635ms
18DOP	$1.13 \cdot 10^7$	$2.14 \cdot 10^6$	$9.39 \cdot 10^5$	$1.21 \cdot 10^6$	423ms	2595ms	3018ms
26DOP	$8.41 \cdot 10^6$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	568ms	2093ms	2661ms

Table 4.2: Benchmark of regular cubes, 12DOP fits the structured mesh nicely.

Compared to the aneurysm example above, the total number of cells is here lower and the number of intersections is higher: 63% of cells in the background mesh are here overlapped, compared to 4% in the aneurysm case. Considering the total time between these cases, it is observed that it is roughly proportional to the number of primitive tests performed. Reducing the number of primitive tests seems in most cases to be the winning strategy; except when comparing 12DOP to 26DOP. Here the number of primitive tests are slightly higher for 12DOP, but the overall time is lower by a small amount. This underlines the dynamic relationship between the tightness of the bounding volume and the cost of its overlap test, as seen in the cost function (4.1).

4.5.4 Benchmark on structured mesh, irregular split cubes

The meshes tested below consists of long and narrow tetrahedrons, constructed to have the same number of cells as the regular cubes example. Long and narrow tetrahedrons are typical in fluid flow simulations where the mesh should resolve the boundary layer.

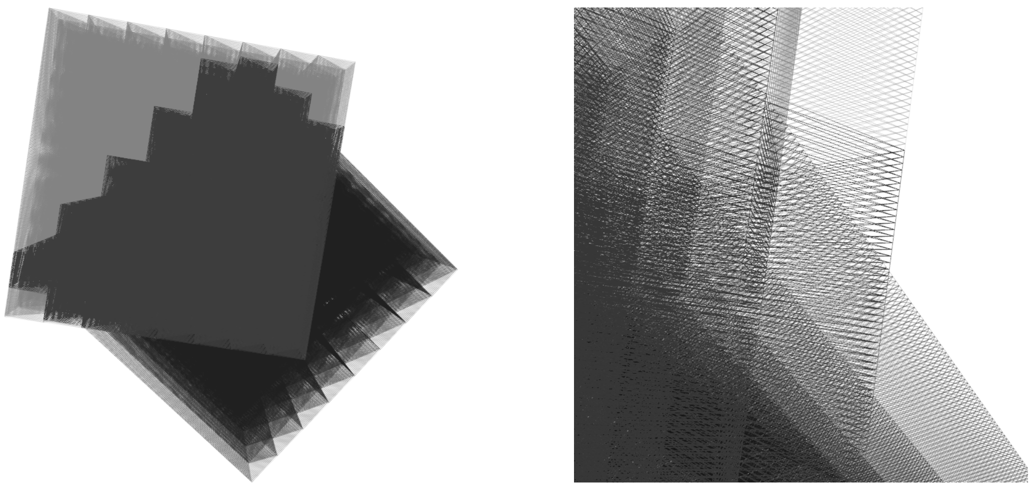

(a) Overview **(b)** Detail

Figure 4.10: Intersections computed on irregular meshes. Cells in background mesh (black): 82944, colliding mesh (light gray): 82944, intersection (mid-tone gray): 55608

Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$2.89 \cdot 10^8$	$1.06 \cdot 10^8$	$7.64 \cdot 10^6$	$9.79 \cdot 10^7$	345ms	103642ms	103987ms
AABB	$7.84 \cdot 10^7$	$3.41 \cdot 10^7$	$7.64 \cdot 10^6$	$2.65 \cdot 10^7$	397ms	38693ms	39090ms
6DOP	$8.61 \cdot 10^7$	$3.41 \cdot 10^7$	$7.64 \cdot 10^6$	$2.65 \cdot 10^7$	378ms	38876ms	39254ms
8DOP	$1.04 \cdot 10^8$	$3.22 \cdot 10^7$	$7.64 \cdot 10^6$	$2.46 \cdot 10^7$	416ms	37586ms	38002ms
12DOP	$4.54 \cdot 10^7$	$1.6 \cdot 10^7$	$7.64 \cdot 10^6$	$8.4 \cdot 10^6$	489ms	21023ms	21512ms
14DOP	$4.91 \cdot 10^7$	$1.66 \cdot 10^7$	$7.64 \cdot 10^6$	$8.94 \cdot 10^6$	445ms	21536ms	21981ms
18DOP	$4.41 \cdot 10^7$	$1.51 \cdot 10^7$	$7.64 \cdot 10^6$	$7.49 \cdot 10^6$	506ms	18345ms	18851ms
26DOP	$3.85 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	466ms	14776ms	15242ms

Table 4.3: Benchmark of irregular cubes, the tightest bounding volume 26DOP performs well.

Although having almost the same number of actual intersections as the regular cubes case, the irregular cells are here queried for collision with a much larger number of candidates. As Table 4.3 shows, the bounding volume hierarchys ability to reduce the number of primitive queries is here paramount for good performance.

4.5.5 Benchmarking with 26DOP at leaf nodes

Motivated by the above benchmarks, tests are here performed with an inserted 26DOP bounding volume at the leaf nodes in the hierarchy. Looking back at Figure 4.5 illustrating the 26DOP bounding volumes in 2D at different levels in the hierarchy, this extra predicate seems to make sense since the actual computed geometry of the 26DOP bounding volumes at intermediate levels, Figures 4.5a, 4.5b, 4.5c, is not in shape far from simpler bounding volumes, like AABBs or 8DOPs. Tables 4.4, 4.5, 4.6 below, in conjunction with Tables 4.1, 4.2, 4.3 above, show that inserting this predicate in most cases reduces the total time spent in the BVH to that of a BVH using a 26DOP.

Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$1.52 \cdot 10^7$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	535ms	795ms	1330ms
AABB	$3.89 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	511ms	512ms	1023ms
6DOP	$4.29 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	503ms	504ms	1007ms
8DOP	$6.89 \cdot 10^6$	$4.75 \cdot 10^5$	$2.45 \cdot 10^5$	$2.3 \cdot 10^5$	533ms	536ms	1069ms
12DOP	$3.02 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	564ms	483ms	1047ms
14DOP	$2.91 \cdot 10^6$	$4.75 \cdot 10^5$	$2.45 \cdot 10^5$	$2.3 \cdot 10^5$	582ms	463ms	1045ms
18DOP	$2.87 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	617ms	485ms	1102ms
26DOP	$2.42 \cdot 10^6$	$4.92 \cdot 10^5$	$2.45 \cdot 10^5$	$2.48 \cdot 10^5$	679ms	501ms	1180ms

Table 4.4: Aneurysm test case with 26DOP bounding volume at leaf nodes.

Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$5.56 \cdot 10^7$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	440ms	3610ms	4050ms
AABB	$1.48 \cdot 10^7$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	433ms	2015ms	2448ms
6DOP	$1.61 \cdot 10^7$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	379ms	1940ms	2319ms
8DOP	$2.11 \cdot 10^7$	$1.36 \cdot 10^6$	$9.39 \cdot 10^5$	$4.19 \cdot 10^5$	394ms	1934ms	2328ms
12DOP	$9.8 \cdot 10^6$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	438ms	1805ms	2243ms
14DOP	$1.06 \cdot 10^7$	$1.36 \cdot 10^6$	$9.39 \cdot 10^5$	$4.19 \cdot 10^5$	438ms	1856ms	2294ms
18DOP	$1.13 \cdot 10^7$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	462ms	1931ms	2393ms
26DOP	$8.41 \cdot 10^6$	$1.41 \cdot 10^6$	$9.39 \cdot 10^5$	$4.75 \cdot 10^5$	531ms	1821ms	2352ms

Table 4.5: Regular cubes test case with 26DOP bounding volume at leaf nodes.

Volume	**Trav's**	**P q's**	true	false	**Constr.**	**Trav.**	**Total**
BSphere	$2.89 \cdot 10^8$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	454ms	21708ms	22162ms
AABB	$7.84 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	407ms	15397ms	15804ms
6DOP	$8.61 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	381ms	15848ms	16229ms
8DOP	$1.04 \cdot 10^8$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.35 \cdot 10^6$	437ms	16810ms	17247ms
12DOP	$4.54 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	447ms	15174ms	15621ms
14DOP	$4.91 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.35 \cdot 10^6$	449ms	15335ms	15784ms
18DOP	$4.41 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	478ms	15509ms	15987ms
26DOP	$3.85 \cdot 10^7$	$1.3 \cdot 10^7$	$7.64 \cdot 10^6$	$5.36 \cdot 10^6$	553ms	15459ms	16012ms

Table 4.6: Irregular cubes test case with 26DOP bounding volume at leaf nodes.

4.5.6 Comparing a BVH to an advancing front algorithm in 2D

The following benchmark compares the implementation of the bounding volume hierarchy with the implementation of the advancing front algorithm. This benchmark serves by no means as an efficiency comparison between these two concepts as a whole. The implementation of bounding volume hierarchy is here employed on 2D triangles lying in the plane, with 3D bounding volumes, primitive tests, construction and traversal strategies. Similarly, the advancing front algorithm is subject to several limitations as described in Section 4.4.2. This benchmark is included as a rudimentary precursor to future development, and is further discussed in Chapter 7.

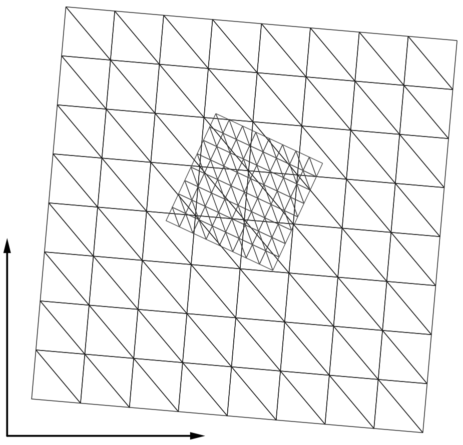


Figure 4.11: The domains used for testing the front advancing algorithm against the bounding volume hierarchy.

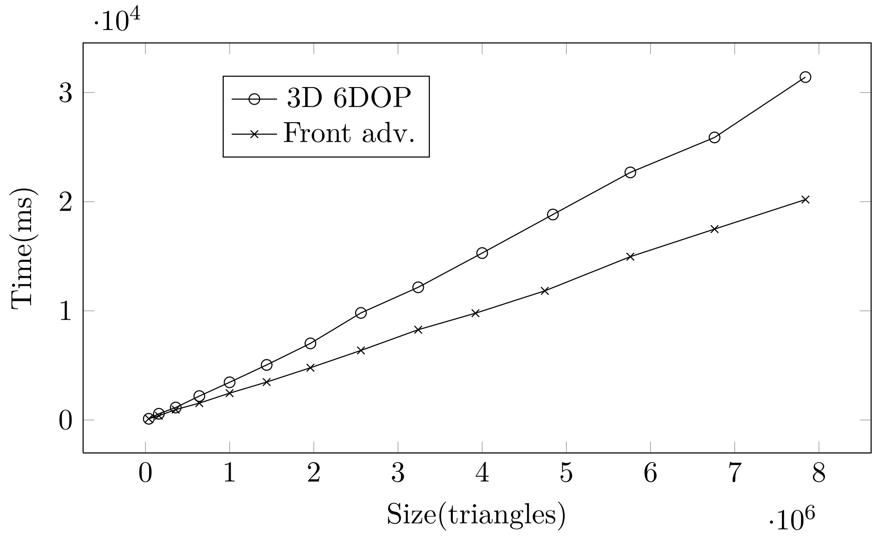


Figure 4.12: Performance of an advancing front algorithm and a bounding volume hiearchy on the domain shown in Figure 4.11. Size is the number of triangles in both meshes combined.

4.6 Mesh decomposition

When an intersection between two meshes has been detected and described, decomposing the domain with respect to this intersection is possible. For the particular application of overlapping meshes, it is necessary to decompose the domain so that integration can be performed on the physical domain $\mathcal{T}_1 \bigcup \mathcal{T}_2$, where $\mathcal{T}_1 = \{T \cap \overline{\Omega}_1 : T \in \mathcal{T}_1^*\}$, as seen in Figure 3.7 in Section 3.4. In Chapter 5 integration will be performed by integrating over a *sub-triangulation* of the cut cells, limited to the case of 2D triangles lying in the plane. Deriving this two dimensional sub-triangulation is described in the following.

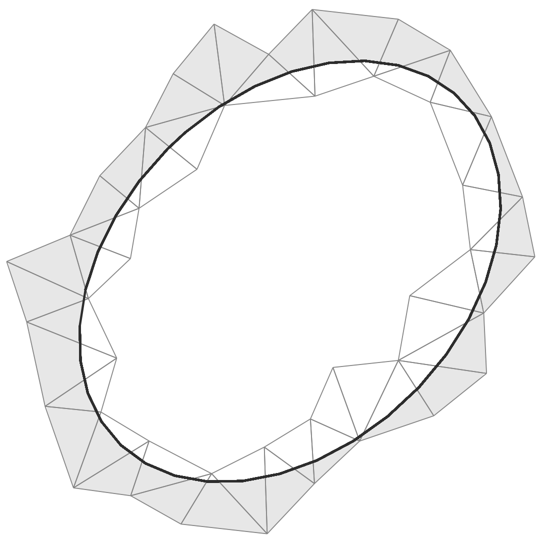


Figure 4.13: Cut triangles within $\mathcal{T}_0$, gray area denote physical part of these, $\mathcal{T}_1 \setminus \mathcal{T}_{0,1}$. This set is found by isolating the triangles overlapped by a triangle in $\mathcal{T}_2$ containing an exterior facet.

4.6.1 Triangulation of cells intersected by an interface

To associated the two domains $\mathcal{T}_1$ and $\mathcal{T}_2$ as described in Section 3.4, the *penalty* term

$$a(u_h, v_h) = (\nabla u_h, \nabla v_h)_\Omega - (\langle \nabla u_h \cdot \mathbf{n} \rangle, [v_h])_\Gamma - ([u_h], \langle \nabla v_h \cdot \mathbf{n} \rangle)_\Gamma + \overbrace{(\gamma h^{-1}[u_h], [v_h])_\Gamma}^{\text{Penalty}}$$

in (3.3) used to weakly enforce the interface condition $u_1 - u_2 = 0$ over the interface Γ. In the case of fitted meshes, this is not a problem, since the triangles all reflect the actual physical domain. To properly impose this interface condition when the interface runs through the domain without necessarily matching the mesh $\mathcal{T}_0$ at its degrees for freedom (here for simplicity assumed to lie on the vertices in $\mathcal{T}_{0,\Gamma}$) shown in Figure 4.13. The function u_1 over cells intersected by Γ needs to reflect the fact that it is not entirely contained in the physical domain. This can done by performing integration just on the parts of the cells contained within Ω_1, motivating the notation of $\mathcal{T}_1$ in Chapter 3.

For overlapping meshes an appropriate triangulation of the cut cells suitable for integration can be found in many ways. One simple strategy is to exploit the fact that when two triangles intersect, their intersection is always a convex polygon. This strategy is tempting because convex polygons are trivial to triangulate; by picking an arbitrary vertex in the polygon and placing edges going from this vertex to all other vertices in the polygon, the polygon is effectively triangulated. The relevant *physical* integral over this cells can then be calculated by first integrating over the whole cut cell and then subtract the contribution from the triangulation of the intersection.

Another strategy to successfully integrate u_1 over the physical part of $T_1 \in \mathcal{T}_{0,\Gamma}$ is to look at the actual concave or convex polygon resulting from T_1 being intersected by the interface Γ. A triangulation of this polygon can be derived by algorithms producing, for an illustrative example, a Delaunay triangulation of the polygon. A triangulation as complex as a Delaunay triangulation would, however, be excessive for the problem at hand. The only purpose of the triangulation of the cut element is to aid the *integration* of the cut cell. The triangles contained within this triangulation is *not in any other way* connected with the conditioning or approximative properties of the linear system (2.9). Thus none of the numerical concerns expressed when meshing a domain in Section 2.2 apply for this triangulation.

The first strategy of subtracting contributions from the convex intersection of two triangles is as mentioned tempting because of its simplicity. This strategy is, however, not easily generalized to properly triangulate the domain if other types of interfaces are running through the domain. An example of this is the fictitious domain case presented in Chapter 3. Here the interface has no belonging triangles needed for this routine, since the interface is only described by a function. Employing the aforementioned strategy would in this case warrant some kind of triangulation of the domain described by the expression defining the interface (or surface) in the fictitious domain case. To retain some flexibility with regards to different descriptions of the interface Γ, a triangulation routine should preferably only rely on a suitable description of the interface itself, and some *inside-outside* information denoting on which side of Γ a function such as u_1 is evaluated.

4.6.2 A greedy triangulation routine

The algorithm presented is that of Bourke [61]. This is a a triangulation routine originally made for polygonising a scalar field and is based on the widely used *marching cubes* algorithm [23]. A similar algorithm is used in LibCutFEM [14] to triangulate surfaces based on *level set* descriptions. The algorithm is here extended to triangulate a polygon intersected by many facet segments. It is in complexity some place in between that of subtracting contributions from a triangulation of the convex intersection between triangles, and that of making a Delanuay triangulation. This algorithm can with little effort be generalized to work on tetrahedrons in 3D. A possible pitfall with this algorithm is that it returns triangulations with a high number of triangles if the initial triangle is intersected by many facets defining the interface. If a triangle in the mesh $\mathcal{T}_0$ is intersected by k facets from $\mathcal{T}_2$, the number of triangles returned is somewhere around $(3^k - k)$. However, since the cells in the different meshes are assumed *compatible* (3.6) over the interface Γ, limiting the number of intersecting facets, this property is not considered a problem.

The algorithm takes a triangle from the triangulation $\mathcal{T}_0$ and a set of border facets from the overlapping domain $\mathcal{T}_2$. A short rundown of the algorithm is as follows:

1. Triangulate initial triangle with respect to the first intersecting facet on the border.
2. Triangulate initial *triangulation* with respect to the second intersecting facet,
3. triangulate the second triangulation with respect to the third intersecting facet and so on...
4. ...until there is no more facets to traverse.
5. Collect inside-outside information and return triangulation outside of $\mathcal{T}_2$.

Algorithm 9 first computes a full triangulation of a triangle T with triangles aligned to the intersecting facets F_i. Then a second pass over the triangulation is done to collect inside-outside information i_T used to remove triangles overlapped by $\mathcal{T}_2$ from the returned triangulation, $\mathcal{T}_{\partial F}$. Obtaining this inside-outside information without doing a more expensive pass on the data prescribes the use of certain functions and overall structure; the use of hyperplanes and *level set* functions within the routine *SingleFacetTriangulation(...)* accomplishes this task.

Algorithm 9 Triangulation routine for a triangle intersected by one or more facets

```
Require: $\triangle(v_0, v_1, v_2), \partial F$
  for each $F_i \in \partial F$ :
      $\mathcal{T}, \mathcal{I}_{\mathcal{T}}$ = SingleFacetTriangulation($v_0, v_1, v_2, F_i, \mathcal{T}, \mathcal{I}_{\mathcal{T}}$)        ▷ Algorithm 10
  for each $i_T \in \mathcal{I}_{\mathcal{T}}$ :
      if $i_T$ contains "1" : $\mathcal{T}_{\partial F}$ += $T$
  return $\mathcal{T}_{\partial F}$
```

Input is the vertices in $\triangle(v_0, v_1, v_2)$, an intersecting facets ∂F. Here $\mathcal{I}_{\mathcal{T}}$ is a set containing aggregated inside-outside information i_T for each triangle $T \in \mathcal{T}$.

Triangulation routine with respect to a single facet

The inner workings of the routine *SingleFacetTriangulation(...)* in Algorithm 9 is now explained. An illustration of the entities and derived data needed for this algorithm can be seen in Figure 4.14 below.

A normal is at first extracted from the border facets of the overlapping triangles, which

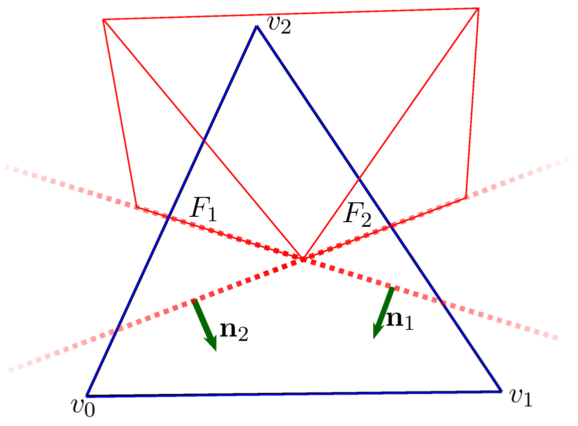


Figure 4.14: Facets, corresponding hyperplanes and normals needed for Algorithm 10. Triangle(blue) is in $\mathcal{T}_0$. Triangles(red) are in $\mathcal{T}_2$, the overlapping mesh.

together with a point on the facet, defines a hyperplane. This hyperplane is used to determine which part of the overlapped triangle is inside or outside the overlapping mesh. To this end we use a *level set* function defined as

$$\phi_{F_i}(\mathbf{x}) = (\mathbf{x} - \mathbf{v}_{F_i}) \cdot \mathbf{n}_{F_i},$$

where $\mathbf{v}_{F_i}$ is a point on a hyperplane aligned with the facet F_i, and $\mathbf{n}_{F_i}$ is the facet normal. This function is negative on one side of the hyperplane and positive on the other, depending on the orientation of the normal $\mathbf{n}_{F_i}$. If the normal has direction outward from the boundary, then this function will be negative outside of $\mathcal{T}_2$ with respect to the particular facet segment F_i.

With the use of $\phi_{F_i}(\mathbf{x})$, the aim is to divide a triangle intersected by a hyperplane into three new triangles sharing amongst them five vertices: three vertices from the original triangle, and two from the intersection of the hyperplane with the edges of our triangle. The location of these two intersections are given by interpolating the edges of the triangle with respect to the level set function $\phi_{F_i}(\mathbf{x})$. This point is found using the interpolating function

$$\mathbf{I}(\mathbf{x}_0, \mathbf{x}_1, F_i) = \mathbf{x}_0 + (\mathbf{x}_0 - \mathbf{x}_1)\frac{\phi_{F_i}(\mathbf{x}_0)}{\phi_{F_i}(\mathbf{x}_1) - \phi_{F_i}(\mathbf{x}_0)}$$

which returns the point of intersection of the facet F_i on the line segment $\overline{(\mathbf{x_0}, \mathbf{x_1})}$.

To triangulate three vertices, each on either side of the hyperplane derived from a facet F_i, there is a total of four cases to consider. To supply each triangle with inside-outside information, this is doubled and a total of eight cases must be considered. Illustrated in Figure 4.15 is the initial triangulation of the example triangle, in Algorithm 10 known as *Case 4*. Here two vertices v_0 and v_1 are outside of the facet F_1, yielding two interpolation points, v_4 on $\overline{v_0 v_2}$ and v_3 on $\overline{v_1 v_2}$, the triangles $\triangle(v_0 v_1 v_4)$, $\triangle(v_1 v_3 v_4)$, $\triangle(v_4 v_3 v_2)$ with labels 1, 1, 0, respectively. Together they constitute the initial labeled triangulation.

Algorithm 10 considers all cases and returns their corresponding triangulation. This algorithm is used on the original triangle and then again for each subsequent facet intersecting the original triangle. For each subsequent call to Algorithm 10, the last triangulation is passed on as input. Proceeding in this fashion gives the subsequent triangulations shown in Figure 4.16. By combining these and removing triangles without the numeral "1" in their label, the triangulation of the area outside $\mathcal{T}_2$ is obtained, as shown in Figure 4.17. At the end of this chapter, a full triangulation of domain $\mathcal{T}_1$ with respect to the exterior of $\mathcal{T}_2$ is given in Figure 4.18.

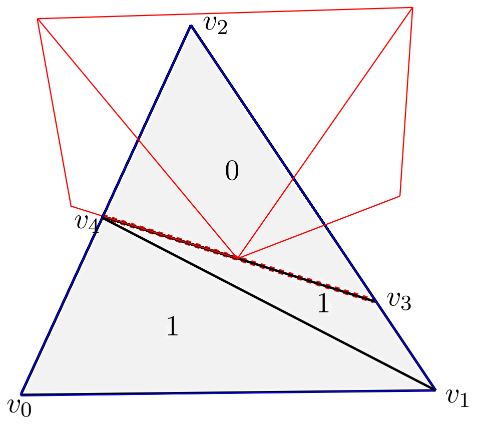


Figure 4.15: Case 4. The initial triangulation of $\triangle(v_0, v_1, v_2)$ wrt. to the hyperplane aligned with F (dashed red line). The points v_3, v_4 are found by $I(v_0, v_2, F)$ and $I(v_1, v_2, F)$.

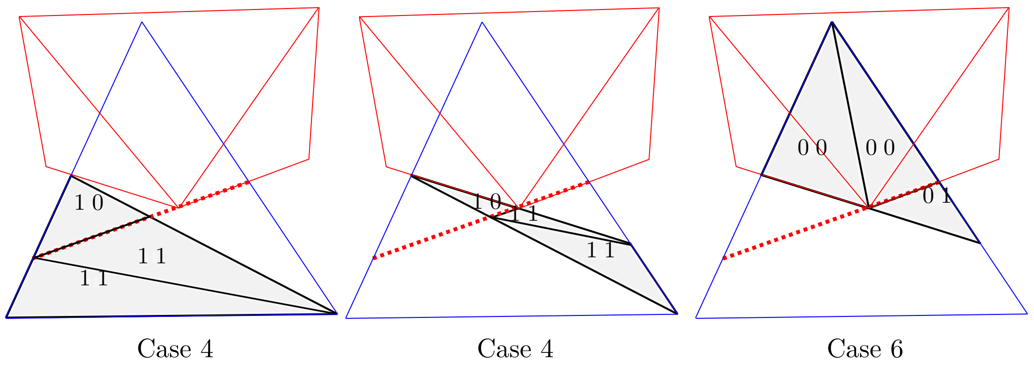


Figure 4.16: Subsequent triangulations of the triangulation shown in Figure 4.15. Note the accumulating inside-outside information i_T: new information corresponding to cases 4, 4, and 6 is appended to information inherited from previous triangulations.

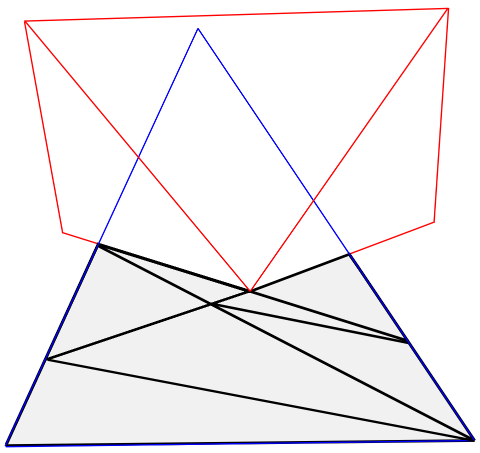


Figure 4.17: All triangles not containing the numeral "1" in i_T are pruned from the combined triangulations, yielding the final triangulation of triangle $\triangle(v_0, v_1, v_2)$ with respect to intersecting facets F_1 and F_2.

Algorithm 10 Triangulation routine for single facet intersecting a triangle

```
Require: $\triangle(v_0, v_1, v_2), F, \mathcal{T}, \mathcal{I}_{\mathcal{T}}$

  if $\phi_F(v_i) < 0$ for $i = 1, 2, 3$ : case $\leftarrow 0$
  if $\phi_F(v_i) > 0$ for $i = 1, 2, 3$ : case $\leftarrow 7$
  if $\phi_F(v_0) < 0$ and $\phi_F(v_1) > 0$ and $\phi_F(v_2) > 0$ : case $\leftarrow 1$
  if $\neg(\phi_F(v_0) < 0$ and $\phi_F(v_1) > 0$ and $\phi_F(v_2) > 0)$ : case $\leftarrow 6$
  if $\phi_F(v_0) > 0$ and $\phi_F(v_1) < 0$ and $\phi_F(v_2) > 0$ : case $\leftarrow 2$
  if $\neg(\phi_F(v_0) > 0$ and $\phi_F(v_1) < 0$ and $\phi_F(v_2) > 0)$ : case $\leftarrow 5$
  if $\phi_F(v_0) < 0$ and $\phi_F(v_1) < 0$ and $\phi_F(v_2) > 0$ : case $\leftarrow 3$
  if $\neg(\phi_F(v_0) < 0$ and $\phi_F(v_1) < 0$ and $\phi_F(v_2) > 0)$ : case $\leftarrow 4$

  if case $= 0$ or case $= 7$ :
      $\mathcal{T}$ += $\triangle(v_0, v_1, v_2)$
      if case $= 0$ : $\mathcal{I}_{\mathcal{T}}$ += "1"
      if case $= 7$ : $\mathcal{I}_{\mathcal{T}}$ += "0"
  if case $= 1$ or case $= 6$ :
      $v_3 \leftarrow \mathbf{I}(v_0, v_1, F)$
      $v_4 \leftarrow \mathbf{I}(v_0, v_2, F)$
      $\mathcal{T}$ += $\triangle(v_0, v_3, v_4)$ += $\triangle(v_3, v_2, v_4)$ += $\triangle(v_3, v_1, v_2)$
      if case $= 1$ : $\mathcal{I}_{\mathcal{T}}$ += "0 1 1"
      if case $= 6$ : $\mathcal{I}_{\mathcal{T}}$ += "1 0 0"
  if case $= 2$ or case $= 5$ :
      $v_3 \leftarrow \mathbf{I}(v_0, v_1, F)$
      $v_4 \leftarrow \mathbf{I}(v_1, v_2, F)$
      $\mathcal{T}$ += $\triangle(v_0, v_3, v_2)$ += $\triangle(v_3, v_4, v_2)$ += $\triangle(v_3, v_1, v_4)$
      if case $= 2$ : $\mathcal{I}_{\mathcal{T}}$ += "1 1 0"
      if case $= 5$ : $\mathcal{I}_{\mathcal{T}}$ += "0 0 1"
  if case $= 3$ or case $= 4$ :
      $v_3 \leftarrow \mathbf{I}(v_1, v_2, F)$
      $v_4 \leftarrow \mathbf{I}(v_0, v_2, F)$
      $\mathcal{T}$ += $\triangle(v_0, v_1, v_4)$ += $\triangle(v_1, v_3, v_4)$ += $\triangle(v_4, v_3, v_2)$
      if case $= 1$ : $\mathcal{I}_{\mathcal{T}}$ += "0 0 1"
      if case $= 6$ : $\mathcal{I}_{\mathcal{T}}$ += "1 1 0"
  return $\mathcal{T}, \mathcal{I}_{\mathcal{T}}$
```

Input is the vertices in $\triangle(v_0, v_1, v_2)$, an intersecting facet F and a set of inside-outside information $\mathcal{I}_{\mathcal{T}}$. Here $\mathcal{I}_{\mathcal{T}}$ += "1 1 0" is an assignment of last added triangle T in $\mathcal{T}$ with inside-outside information. "0" for the last, "1" for the second last and "1" for the third last.

4.6.3 Implementation of the triangulation algorithm

The code for the implementation that follows can be found at:
`https://github.com/tomana/compgeom_olm/tree/master/tesselation`

Verifying the implementation

An application specific verification of this algorithm is straight forward: if the algorithm does its prescribed job, the area of the computed $\mathcal{T}_1$ (shown in Figure 3.7, Section 3) simply equals the area of $\Omega_1 - \Omega_2$. This can shown to be the case for all the meshes considered in this thesis. Code for doing this test can be found in A.2. This verification, although by no means completely watertight, can serves as a rudimentary verification of all the above steps related to collision detection before using the domains in numerical simulations.

4.6.4 Data structures

A set of handy data structures is needed to successfully assemble a system of equations on overlapping meshes. The assembly algorithm presented in the coming Chapter 5 needs access to the different entities returned from the computational domain decomposition described above. To this end a typical mesh data structure is used to store triangulations of cut cells, and this container is further enriched with maps denoting the relevant connectivity information between the derived cut entities and original input meshes $\mathcal{T}_0$ and $\mathcal{T}_2$.

For integration on the physical part of $\mathcal{T}_0$, $\mathcal{T}_1$, a container for the local triangulated area outside Γ and a map associating each triangle in $\mathcal{T}_{0,\Gamma}$ with its this triangulation is used. The set containing the local triangulations for all the triangles in $\mathcal{T}_{0,\Gamma}$ is here denoted

$$\mathcal{T}_{1,\text{cut}}.$$

The triangulations contained within this set is illustrated in the upper part of Figure 4.19 located at the end of this chapter. Here a triangulation (marked $\triangle$) of a triangle with index 3 in the background mesh is shown. The *parent-entity map*

$$\mathcal{P}_{1,\text{cut}} : \{i_p : T_i \in \mathcal{T}_{1,cut}\},$$

associates each triangle in the set $\mathcal{T}_{1,cut}$ with its *parent-entity*, which for the local triangulation marked $\triangle$ in Figure 4.19 is $i_p = 3$, the triangle with index 3 in $\mathcal{T}_0$.

To associate the interface with the relevant cells in the different domains, a preliminary step is to partition each facet on Γ into a set of polygons $\{\Gamma_{kl}\}$ such that each polygon Γ_{kl} intersects exactly one cell $T_{2,k}$ of the overlapping mesh $\mathcal{T}_2$ and one cell $T_{0,l}$ of the background mesh $\mathcal{T}_0$. This process is naturally also subject to all the numerical concerns expressed so far. The code for this can be found in the implementation [62], where it should be noted that this particular code has proven to be error prone in some limited cases. Simple as it seems, cases resulting in an ambiguity between an algorithm computing the intersection of a point on a line and an algorithm detecting the intersection of a point with a triangle, sometimes arises here.

For the segmented interface facets $\{\Gamma_{kl}\}$ there are two maps: one denoting a parent element in $\mathcal{T}_0$ and the other a parent element in $\mathcal{T}_2$. The collection $\{\Gamma_{kl}\}$ is denoted

$$\mathcal{T}_{\Gamma,\text{cut}},$$

where the triangulation notation "$\mathcal{T}$" is used to reflect the fact that in a similar three dimensional case, the interface would be triangulation of the surface of $\mathcal{T}_2$. In Figure 4.19 the segments contained within this set is marked $\angle$. The parent-entity map

$$\mathcal{P}_{\Gamma,\text{cut}} : \{(i_p, j_p) : F_i \in \mathcal{T}_{\Gamma,\text{cut}}\}$$

associates each facet segments in the set $\mathcal{T}_{\Gamma,\text{cut}}$ with a two parent entities in the meshes $\mathcal{T}_{0,\Gamma}$ and $\mathcal{T}_2$, respectively.

In the implementation [62] the parent entity maps are put in a specific container of type `std::unordered_map`, also seen in the code for the bounding volume hierarchy. The key-value pairs are here so that the keys correspond to a triangle or facet segment in the sets $\mathcal{T}_{1,\text{cut}}$ and $\mathcal{T}_{\Gamma,\text{cut}}$. For $\mathcal{P}_{1,\text{cut}}$ the values correspond to a triangle in $\mathcal{T}_0$. For $\mathcal{P}_{\Gamma,\text{cut}}$ the values are two triangles, the first belonging to the background mesh and the second belonging to the overlapping mesh. The collections of triangulations and segmentations are stored in two separate `dolfin::Mesh`'es in the Dolfin [54] framework, one for $\mathcal{T}_{1,\text{cut}}$ and one for $\mathcal{T}_{\Gamma,\text{cut}}$. One mesh and the belonging parent entity map(s) are put into a class `cutfem::CutMesh` contained within LibCutFEM [14]. The LibCutFEM library is introduced and extensively used in the coming Chapters 5 and 6.

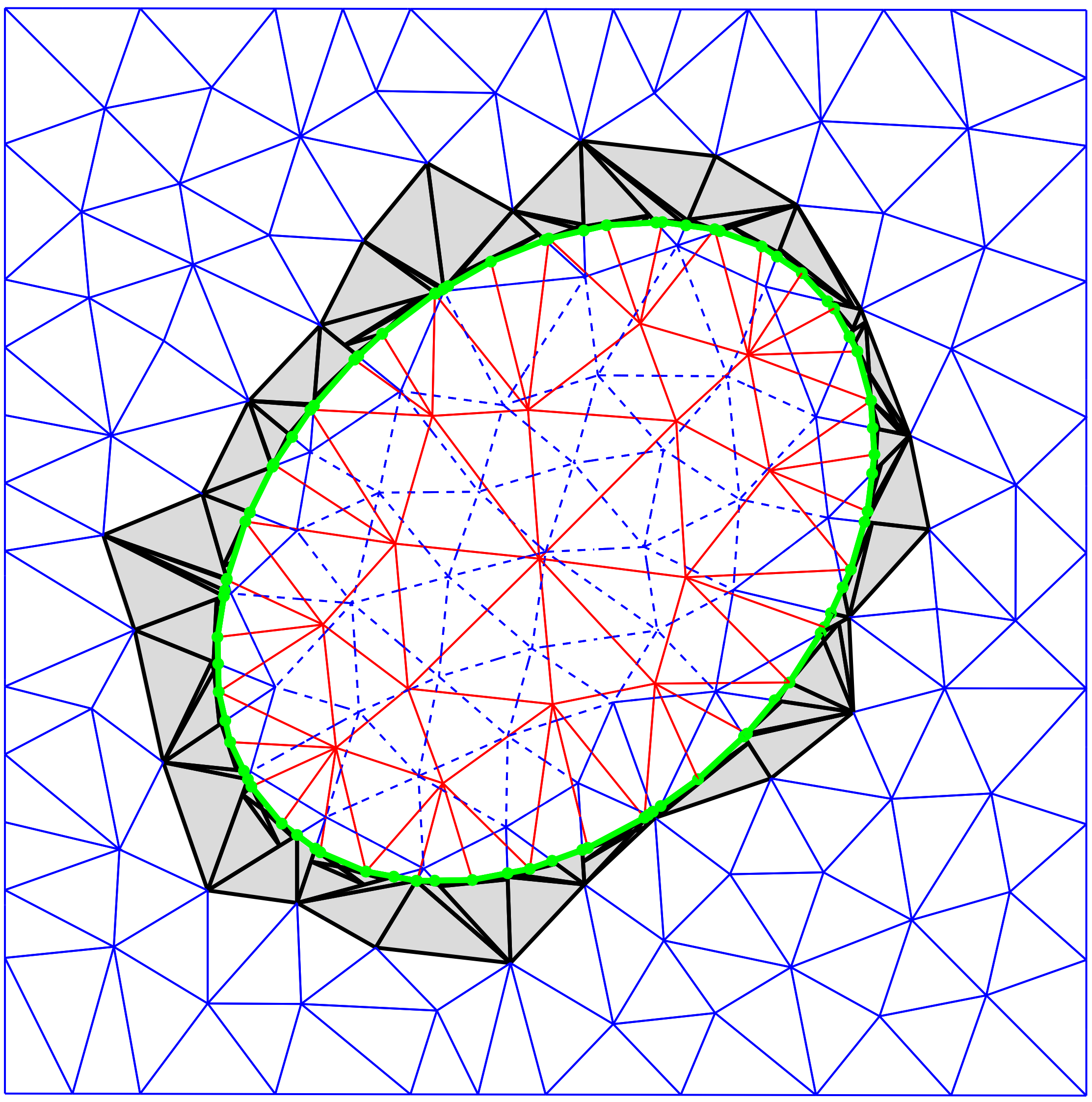


Figure 4.18: Complete triangulation of domain $\mathcal{T}_1$ (blue outer triangles plus black triangles with gray interior) with respect to the exterior of $\mathcal{T}_2$ (red triangles). Green segments denote the segmented interface Γ. Dashed lines denote elements of $\mathcal{T}_0$ that are completely overlapped by $\mathcal{T}_2$ (red triangles).

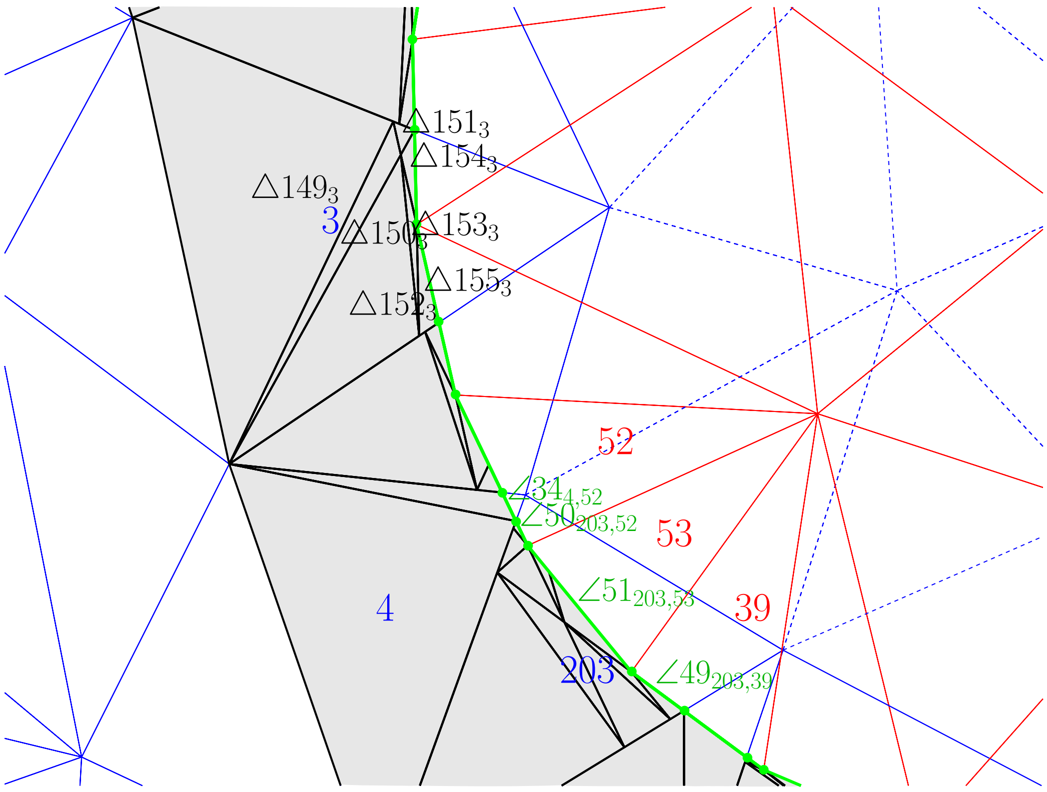


Figure 4.19: Zoom-in on lower part of Figure 4.18. Here entity indexes and subscripts illustrate data structures. Gray area is $\mathcal{T}_{1,\text{cut}}$. A particular triangulation contained in $\mathcal{T}_{1,\text{cut}}$ (triangles marked $\triangle$) has subscripts denoting its parent entity in $\mathcal{T}_0$. A part of the cut interface $\mathcal{T}_{\Gamma,\text{cut}}$ (facet segments marked $\angle$) is shown with subscripts denoting parent entities in $\mathcal{T}_0$ and $\mathcal{T}_2$.

Chapter 5

Integration and assembly of cut elements

As mentioned in Section 3.5.2, the standard approach of applying Gaussian quadrature rules over the cells in a domain is not directly applicable on a domain consisting of cut elements like those arising from from superimposing $\mathcal{T}_2$ on $\mathcal{T}_0$.

In Section 3.4 the triangulation $\mathcal{T}_0$ of the background domain Ω_0 is decomposed into three disjoint subsets:

$$\mathcal{T}_0 = \mathcal{T}_{0,1} \cup \mathcal{T}_{0,2} \cup \mathcal{T}_{0,\Gamma}$$

To assemble the linear system needed to solve a problem on overlapping meshes, each subset in this domain is treated individually. The first two subsets are readily handled; a standard assembly algorithm can be used to assemble the integrals over the cells in $\mathcal{T}_{0,1}$ and integrals over the cells of $\mathcal{T}_{0,2}$ need not be assembled at all since these are completely overlapped by the domain $\mathcal{T}_2$. Integrating on cells over the domain $\mathcal{T}_{0,\Gamma}$, the partially overlapped cells of $\mathcal{T}_0$, however, requires additional machinery since only parts of the cells in $\mathcal{T}_{0,\Gamma}$ is contained in the physical domain. Furthermore, the interface Γ contained in this domain requires special treatment to properly couple Ω_1 and Ω_2.

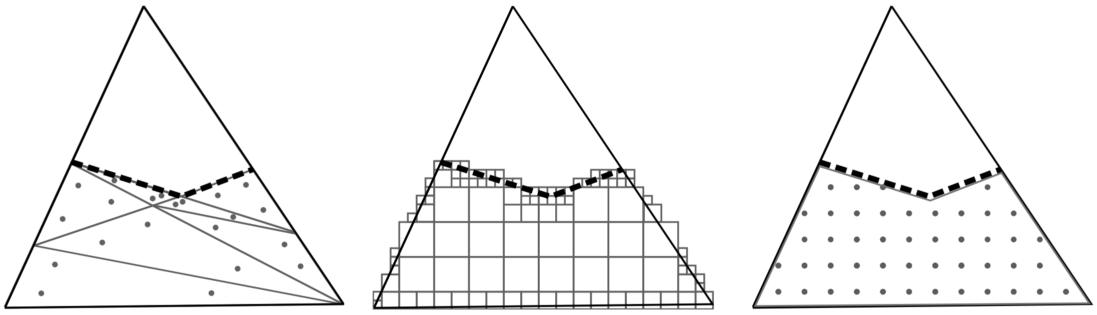
When using the Nitsche method on interface problems, different terms in the weak formulation are used to enforce compliance along the interface Γ. For the model problem (3.3) these terms are

$$a(u_h, v_h) = (\nabla u_h, \nabla v_h)_\Omega - \overbrace{(\langle \nabla u_h \cdot \mathbf{n} \rangle, [v_h])_\Gamma}^{\text{Consistency}} - \overbrace{([u_h], \langle \nabla v_h \cdot \mathbf{n} \rangle)_\Gamma}^{\text{Symmetry}} + \overbrace{\gamma h^{-1}([u_h], [v_h])_\Gamma}^{\text{Penalty}},$$

where the integrands involve products of trial and test functions defined on different meshes. When $\mathcal{T}_2$ is completely contained within $\mathcal{T}_1$, then interface Γ consists of the boundary facets $\partial_e \mathcal{T}_2$ of $\mathcal{T}_2$. Initially these facets may intersect several cells of $\mathcal{T}_0$, and a preliminary step is to decompose the facets into segments contained within only one cell from $\mathcal{T}_0$, as mentioned in Section 4.6.4.

In addition to coupling the domains over the interface Γ, integrating on the partially overlapped cells of $\mathcal{T}_0$ has to be performed. Before an assembly algorithm is presented at the end of this chapter, some ways to integrate on cut elements are review. A good review of many other integration techniques for convex and concave volumes is given in Sudhakar [63].

5.1 A summary of integration techniques for cut elements



(a) Tesselation approach; original cell is tesselated along the interface cutting the cell, quadrature points (dots) from standard quadrature rules placed in the resulting triangles.

(b) Adaptive approach; successive refinement of *integration cells* (squares) along interface cutting element.

(c) Quadrature points (dots) placed and weighted according to moment fitting equations in polygon resulting from intersection by interface.

Figure 5.1: Different approaches to integration on cut elements.

Triangulating the polygon resulting from T_l^0 being cut by Γ and doing integration over the triangles contained within this triangulation [64, 65, 66, 67] is perhaps the most intuitive and practical approach to integration on cut elements. A framework developed for the finite element method already contain much of the needed machinery for this approach, like quadrature rules for triangles and tetrahedrons. The idea as illustrated in Figure 5.1a is to perform integration on the cut element by collecting the quadrature points available on a subtriangulation of the cut cell, where this subtriangulation is aligned with the interface cutting the cell. Some possible disadvantages of this an approach lies in the triangulation algorithms. The triangulation algorithm presented in Section 4.6 is for instance limited if it is intersected by many facets of the interface Γ.

Decomposing the domain into cells that conform to the interface Γ is also possible in a more homogeneous fashion, via an *adaptive quadrature approach* [68, 69], illustrated in Figure 5.1b. The polygon resulting from the intersection with Γ is here split into integration cells that are recursively refined until the error in integration between successive refinements is below a certain threshold. In this approach the *integration cells* are not necessarily perfectly aligned with the interface, introducing another source of error in the approximation. Figure 5.1b illustrates this approach with square cells. The number of integration cells needed for this approach is high, but arguably more predictable than with the triangulation approach outlined above.

Yet another approach is to use *moment fitting* equations [70, 13] to derive quadrature points and weights, which gives rise to a multitude of quadrature rules suitable for many purposes. Where a *moment* denotes some kind of a quantitative measure of a set of points. A simple example of such a moment is using the *center of gravity* of a polygon given by

$$\mathbf{x}_c = \frac{1}{|P|} \int_P \mathbf{x} \, \mathrm{d}V(\mathbf{x}),$$

as a quadrature point and letting the corresponding quadrature weight be the volume or area $|P|$ of the intersected cell $P = T \cap \Omega_1$. More complex rules based on moment fitting equations can be seen in Sudhakar [63, 71], where techniques such as *ray-shooting*, commonly associated with rendering and computer graphics, are used to find candidates for quadrature points within polygons of arbitrary shape.

5.2 Assembly of the linear system

Algorithm 11 constructs a global matrix for the overlapping mesh case. This algorithm is here using, but is not limited to, the triangulations of the physical part of cut entities $\mathcal{T}_{1,cut}$ as the domain of integration when assembling cells over $\mathcal{T}_{0,\Gamma}$. The local basis functions needed for integration over an interface segment $\tilde{F} \in \mathcal{T}_{\Gamma,\mathrm{cut}}$ cutting a triangle $T_1 \in \mathcal{T}_{0,\Gamma}$ and itself belonging to a triangle $T_2 \in \mathcal{T}_2$ are

$$\phi_i^{(\tilde{T})} = \begin{cases} \phi_i^{(T_1)} & 0 \leq i \leq \dim(V_1(T_1)) \\ \phi_{i+\dim(V_1(T_1))}^{(T_2)} & \dim(V_1(T_1)) < i \leq \dim(V_1(T_1)) + \dim(V_2(T_2)), \end{cases} \tag{5.1}$$

where $\phi_i^{(T_1)}$ belongs to the local function space over T_1 and $\phi_{i+\dim(V_1(T_1))}^{(T_2)}$ belonging to T_2 with its indexes *shifted* by the number of local basis functions in $V_1(T_1)$. Further, the notation $\mathcal{I}(\cdot)$ in Algorithm 11 denotes an index-set: denoting either the index of a local basis function over a triangle, or in the cases $\mathcal{I}(\tilde{F})$ and $\mathcal{I}(\tilde{T})$, the index of a facet segment contained in $\mathcal{P}_{\Gamma,cut}$ or triangle in $\mathcal{P}_{1,cut}$, as described in Section 4.6.4. The assignment $(i_p, j_p) = \mathcal{P}_{\Gamma,cut}(\mathcal{I}(\tilde{F}))$ here specifically means extracting the index i_p of a triangle T_i in $\mathcal{T}_0$, and index j_p of a triangle T_j in $\mathcal{T}_2$. The local indices r and s are mapped to their corresponding global indices by a local-to-global mapping $q(\tilde{T}, r) = (q_1(T_1, r), q_2(T_2, r))$, where q_1 is the local to global mapping corresponding to V_1 and q_2 the shifted local to global mapping belonging to V_2. The function *QuadratureRule*(...) either extracts the quadrature rule Q defined on a single triangle, or a rule defined on two triangles associated with a facet segment on the interface Γ.

The notation $a^Q(\cdot,\cdot)$ means performing numerical integrating of a bilinear form using the quadrature rule Q as extracted prior.

Algorithm 11 Assembly of global matrix

```
$A = 0$
**for each** $T \in \mathcal{T}_{0,1} \cup \mathcal{T}_2$ :
    $\mathcal{I}(T) = \{1, ..., dim(V_1(T))\} \times \{1, ..., dim(V_1(T))\}$
    **for each** $(r,s) \in \mathcal{I}(T)$ :
        $A^{(T)}_{r,s} = a(\phi^{(T)}_r, \phi^{(T)}_s)$
    **for each** $(r,s) \in \mathcal{I}(T)$ :
        $A_{q(T,s),q(T,r)} += A^{(T)}_{r,s}$
**for each** $\tilde{T} \in \mathcal{T}_{1,\mathrm{cut}}$ :
    $i_p = \mathcal{P}_{1,cut}(\mathcal{I}(\tilde{T}))$
    $T = T_{i_p} \in \mathcal{T}_{0,\Gamma}$
    $\mathcal{I}(\tilde{T}) = \{1, ..., dim(V_1(T))\} \times \{1, ..., dim(V_1(T))\}$
    $Q = \text{QuadratureRule}(T)$
    **for each** $(r,s) \in \mathcal{I}(\tilde{T})$ :
        $A^{T}_{r,s} = a^Q(\phi^{(T)}_r, \phi^{(T)}_s)$                ▷ Integrate using the quadrature rule $Q$.
    **for each** $(r,s) \in \mathcal{I}(\tilde{T})$ :
        $A_{q(T,s),q(T,r)} += A^{(\tilde{T})}_{r,s}$
**for each** $\tilde{F} \in \mathcal{T}_{\Gamma,\mathrm{cut}}$ :
    $(i_p, j_p) = \mathcal{P}_{\Gamma,cut}(\mathcal{I}(\tilde{F}))$
    $T_1 = T_{i_p} \in \mathcal{T}_{0,\Gamma}$
    $T_2 = T_{j_p} \in \mathcal{T}_2$
    $\mathcal{I}(\tilde{T}) = \{1, ..., dim(V_1(T_1))\} \times \{1, ..., dim(V_2(T_2))\}$       ▷ !!!!!
    $Q = \text{QuadratureRule}((T_1, T_2))$                ▷ Quadrature rule over the interface.
    **for each** $(r,s) \in \mathcal{I}(\tilde{T})$ :
        $A^{T}_{r,s} = a^Q(\phi^{(\tilde{T})}_r, \phi^{(\tilde{T})}_s)$                ▷ This uses the local basis functions (5.1).
    **for each** $(r,s) \in \mathcal{I}(\tilde{T})$ :
        $A_{q(T,s),q(T,r)} += A^{(\tilde{T})}_{r,s}$
```

5.3 Implementation in FEniCS and LibCutFEM

Algorithm 11 is implemented in LibCutFEM [14], a framework for automated assembly of general cut finite element based variational forms over fictitious domains and other unfitted geometries within Dolfin [54] and is developed by Massing [12, 13, 36], Claus [72, 73] and others.

Chapter 6

Numerical results

6.1 Poisson on overlapping meshes

6.1.1 Formulation

For easy reference, the problem given in Chapter 3 is here given

$$\begin{aligned} -\Delta u_1 &= f_1 && \text{in } \Omega_1, \\ -\Delta u_2 &= f_2 && \text{in } \Omega_2, \\ [\nabla u \cdot \mathbf{n}] &= 0 && \text{on } \Gamma, \\ [u] &= 0 && \text{on } \Gamma, \\ u &= g && \text{on } \partial\Omega_{0,D}, \\ \nabla u \cdot \mathbf{n} &= 0 && \text{on } \partial\Omega_{0,N} \end{aligned} \tag{6.1}$$

where $\mathbf{n}$ is the unit normal directed from Ω_1 into Ω_2, and $[u] = u_1 - u_2$ denotes a jump over the interface Γ.

The discretized weak formulation for this problem using the Nitsche method is

$$\begin{aligned} a(u, v_h) =& (\nabla u_h, \nabla v_h)_{\Omega_1 \cup \Omega_2} - (\langle \nabla u_h \cdot \mathbf{n} \rangle, [v_h])_\Gamma - ([u_h], \langle \nabla v_h \cdot \mathbf{n} \rangle)_\Gamma + \gamma h^{-1}([u_h], [v_h])_\Gamma \\ & - (\nabla u \cdot \mathbf{n}, v_h)_{\partial\Omega} - (\nabla v_h \cdot \mathbf{n}, u_h)_{\partial\Omega} + \gamma h^{-1}(u_h, v_h)_{\partial\Omega} \\ l(v_h) =& (f_h, v_h)_\Omega - (\nabla v_h \cdot \mathbf{n}, g)_{\partial\Omega} + \gamma h^{-1}(g, v_h)_{\partial\Omega} \end{aligned}$$

where both a weak enforcement boundary and interface conditions have been employed, as explained in Sections 3.2.1 and Section 3.3. And $\alpha_1 = 0$ and $\alpha_2 = 1$ for the normal fluxes across the interface (3.5). The *source* functions are chosen to be

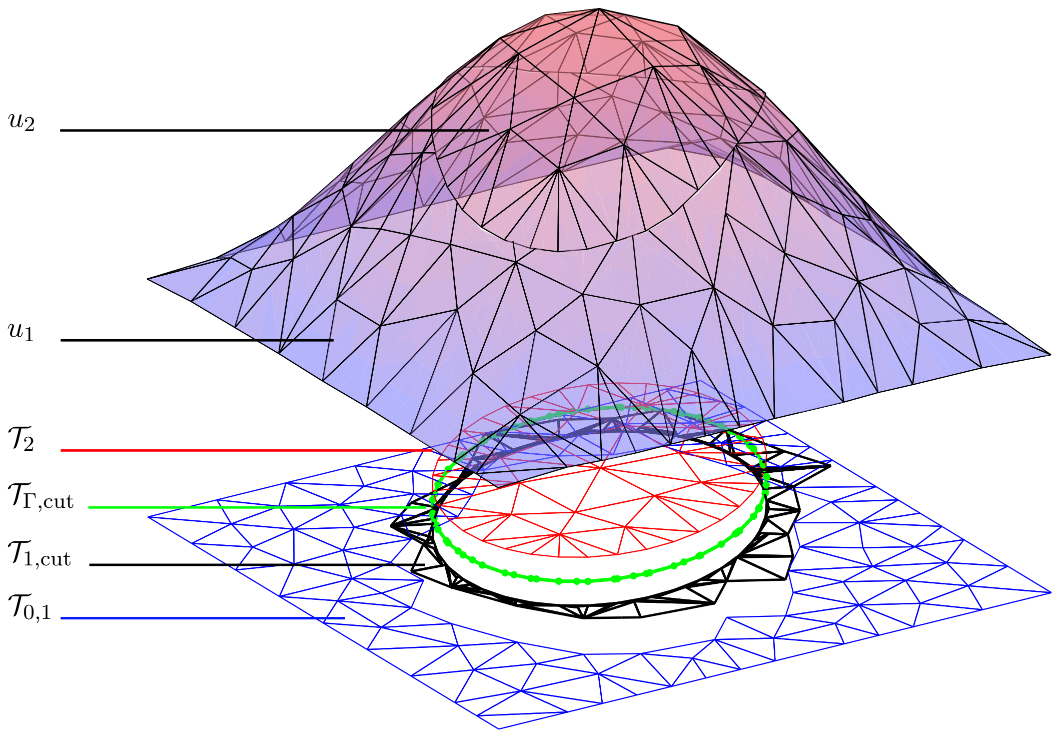
$$f_1(x, y) = f_2(x, y) = 2\pi^2(\sin(\pi x)\sin(\pi y)). \tag{6.2}$$

and $g = 0$.

6.1.2 Implementation

The solution algorithm is made using implementations of algorithms described in Chapter 4 and LibCutFEM [14]. Code for solving this in the FEniCS framework with LibCutFEM and the geometrical tools available can be found in A.3. In the first case, illustrated in Figure 6.1, P_1-elements have been used and the parameter γ is set to 10. The quadrature rule used on the domains $\mathcal{T}_{0,1}$ and $\mathcal{T}_2$ is of order 1 and on the elements contained in $\mathcal{T}_{0,\Gamma}$ it is of order 2.. To verify the solution algorithms, a convergence test and *patch test* is performed in the coming sections.

6.1.3 Result



u_2
u_1
$\mathcal{T}_2$
$\mathcal{T}_{\Gamma,\text{cut}}$
$\mathcal{T}_{1,\text{cut}}$
$\mathcal{T}_{0,1}$

Figure 6.1: Solution of the Poisson's equation on overlapping meshes $\mathcal{T}_0$ and $\mathcal{T}_2$, shown with derived geometrical entities. The solution u_0 has been interpolated over the cut triangles in $\mathcal{T}_{0,\Gamma}$ to reflect the solution on the physical part of this domain, $\mathcal{T}_{1,\text{cut}}$. The boundary condition is weakly enforced as described in Section 3.2.1 with $\gamma = 10$.

6.1.4 Convergence test

With the source functions f_1 and f_2 as given in (6.2), the problem (6.1) has the analytical solutions

$$u_1(x, y) = u_2(x, y) = \sin(\pi x) \sin(\pi y).$$

This solution is interpolated over the domains and compared to the approximated solution. The error measure is $u_{\text{error}} = ||u_{\text{exact}} - u_h||_{\mathrm{H}_1}$, using the H^1 norm (2.4), and the *estimated order of convergence* (EOC) of u_h is here defined as

$$\mathrm{EOC}(k) = \frac{\log u^{k}_{\text{error}} - \log u^{k-1}_{\text{error}}}{\log h^{k}_{max} - \log h^{k-1}_{max}}.$$

In Tables 6.1 6.2 the estimated order of convergence and error in H^1 as they appear when using the parameterized mesh shown in Figure 6.2 with generation code in appendix, A.7.

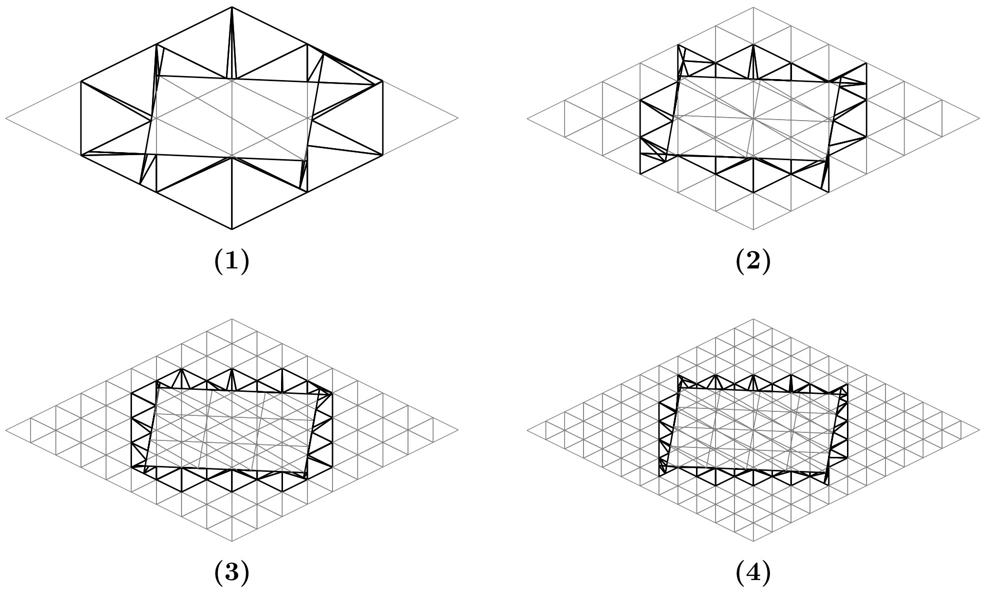
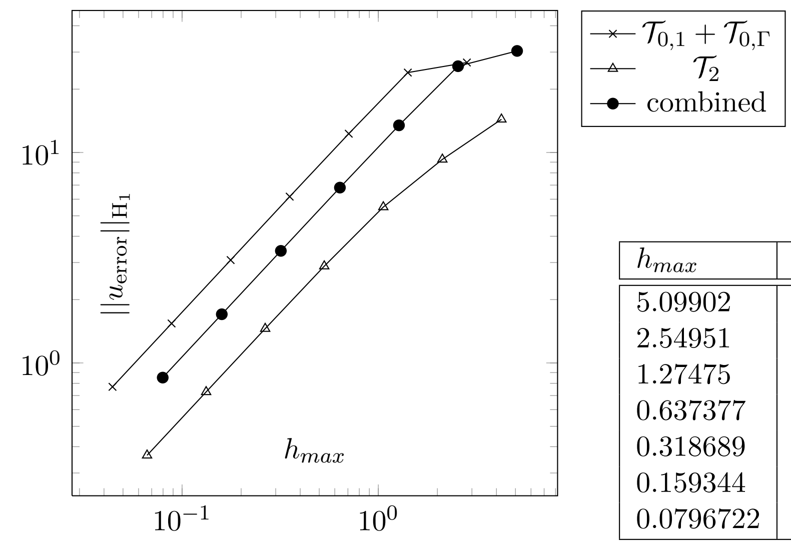


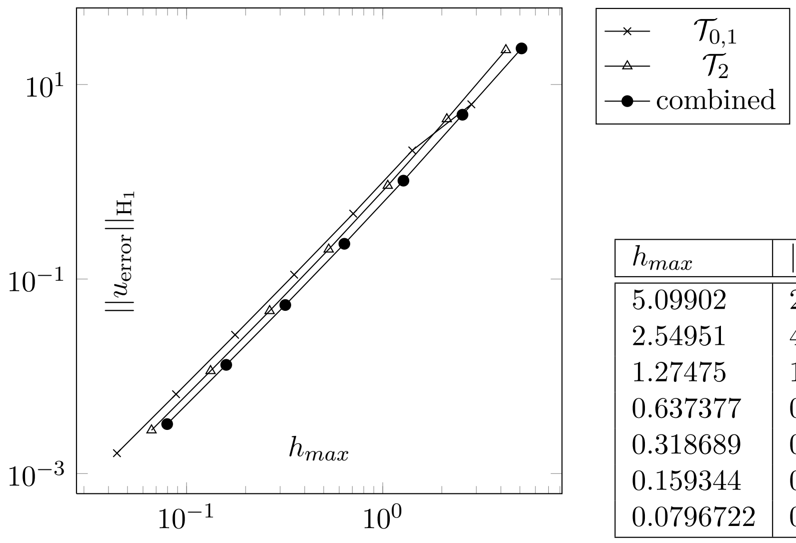
Figure 6.2: Plot of parameterized meshes. Number of elements in background of mesh is $(3(2^i))^2$ for $i = 0, 1, ..., 7$ and number of elements in of overlapping mesh is $(2^i)^2$ for $i = 0, 1, ..., 7$, respectively. Note how the triangles in the triangulation of $\mathcal{T}_{1,\text{cut}}$, denoted by black thick lines, are different in each refinement.



h_{max}	$\|\|u_{\text{error}}\|\|_{\mathrm{H}_1}$	EOC
5.09902	30.3786	
2.54951	25.7026	0.241141
1.27475	13.4585	0.933398
0.637377	6.80976	0.982843
0.318689	3.40822	0.998585
0.159344	1.70368	1.00037
0.0796722	0.851815	1.00004

Figure 6.3: Plot of the error $||u_{\text{error}}||_{\mathrm{H}_1}$ for of Poisson's equation using P_1 elements.

Table 6.1: Estimated order of convergence and error for P_1 elements.



h_{max}	$\|\| \cdot \|\|_{\mathrm{H}_1}$	EOC
5.09902	23.5048	
2.54951	4.88905	2.26533
1.27475	1.02667	2.25158
0.637377	0.230324	2.15624
0.318689	0.0540704	2.09075
0.159344	0.0131114	2.04402
0.0796722	0.0032264	2.02282

Figure 6.4: Plot of the error $||u_{\text{error}}||_{\mathrm{H}_1}$ for of Poisson's equation using P_2 elements.

Table 6.2: Estimated order of convergence and error for P_2 elements.

6.1.5 Patch test

For P_1 elements, setting $u = 1 + x + y$ gives us $-\Delta(u) = 0$. Setting $f_1 = f_2 = 0$ and $g = 1 + x + y$ in (6.1). Since these polynomials are of first order, it is expected that the finite element method will return the exact solution. This is shown to be the case, up to error $8.97552e - 14$ in the H_1 norm for P_1 elements. Setting $f_1 = f_2 = -4$ and

$g = x^2 + y^2 + xy + x + y + 1$ the expected solution is $x^2 + y^2 + xy + x + y + 1$, and the error is $6.36258e - 12$ for P_2 elements in the mesh shown in Figure 6.1. The less accurate result for P_2 elements shown here is probably attributed to the fact that the elements over the interface Γ was integrated using a quadrature rule of order 3 due to the availability of quadrature rules in the finite element framework when these tests were performed. A quadrature rule of order 4 should be employed to properly integrate the basis functions in the broken function space $P_1^*(\mathcal{T}_1)$ as illustrated in Figure 3.8 when the original elements are P_2 elements.

6.1.6 Timings

The time spent computing the different parts involved in this simulation – collision detection, mesh decomposition and assembly and solving – are shown in Figures 6.5 and 6.6. In the first plot these timings are separate, and in the second plot, the timings are stacked on top of each other to reflect the timing of the whole process. These simulations have been performed on the parameterized meshes illustrated in Figure 6.2. For good measure a solution of the Poisson's equation, with strongly enforced boundary conditions and the same source function on the mesh $\mathcal{T}_0$, is included in both plots.

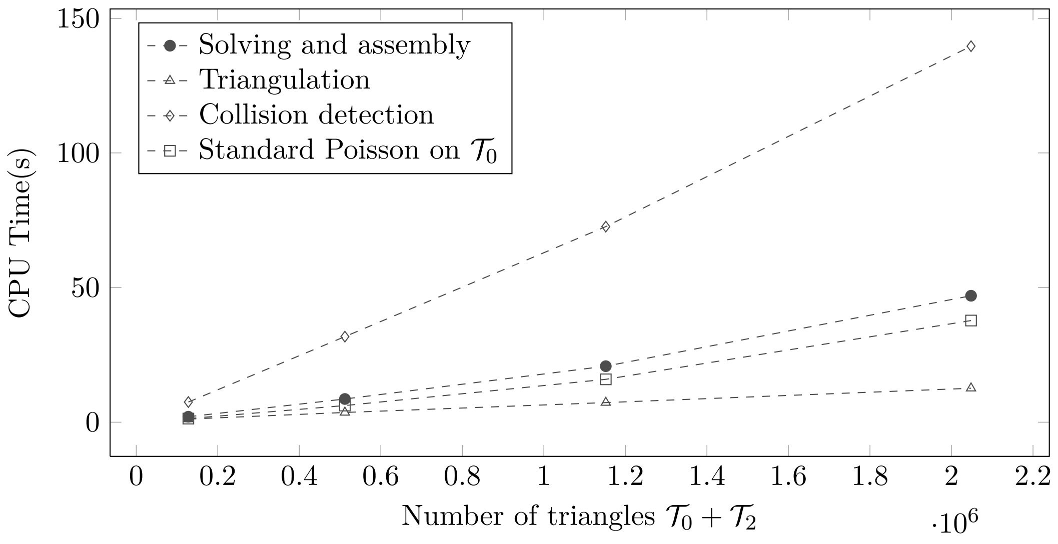
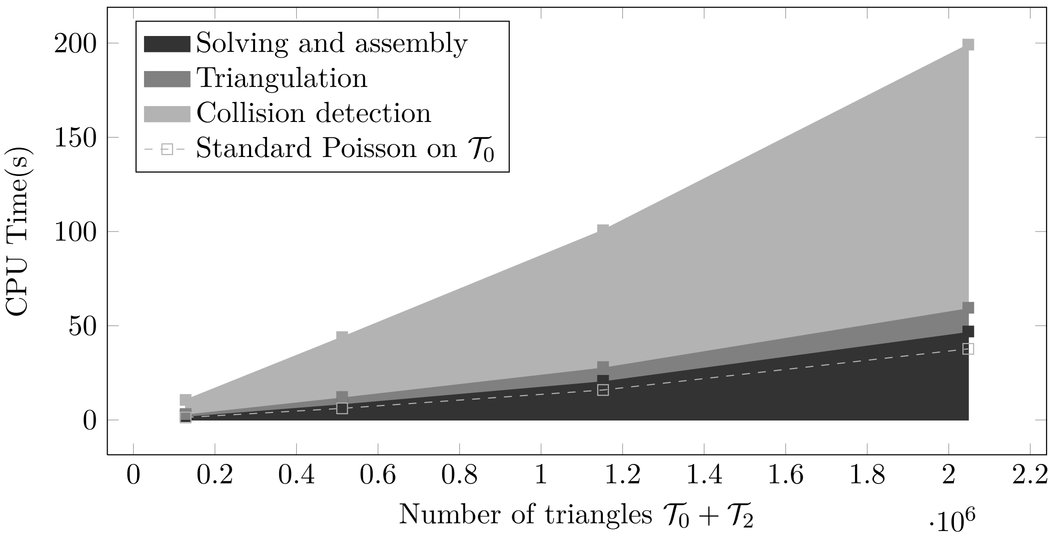


Figure 6.5: Linear plot of CPU time(s) showing computing time for different algorithms given two meshes $\mathcal{T}_0$ and $\mathcal{T}_2$.



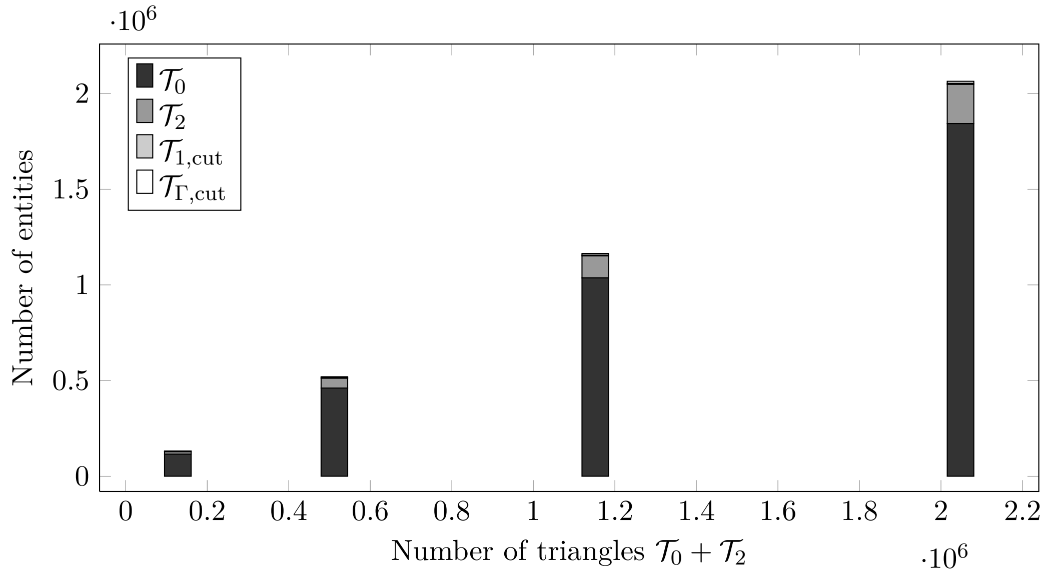


Figure 6.6: (Top) Linear plot of CPU time(s) vs. mesh size, showing a stacked plot yielding total time given two meshes $\mathcal{T}_0$ and $\mathcal{T}_2$. Included is a plot of the solution of a standard Poisson problem formulation on $\mathcal{T}_0$. (Bottom) Number of entities in different meshes, triangles or line segments.

Chapter 7

Conclusions and further work

7.1 Computational geometry

This thesis has followed the development of several algorithms made with the intent of geometrically decomposing a meshed domain into suitable entities for finite element analysis via the Nitsche method on overlapping meshes. As a precursor to further development of such a geometrical toolbox this thesis provides the insights given in the following sections.

7.1.1 Bounding volume hierarchy

In Section 4.5.1 it is shown that when using a bounding volume hierarchy to perform intersection detection on meshes typical to finite element analysis in three dimensions, a *tight* bounding volume enclosing single primitives (at leaf nodes) is important for computational efficiency. A tight bounding volume both reduces overall traversal time and limits worst case behavior associated with irregular meshes shown in Section 4.5.4. Furthermore, it is shown that this tightness is not as important at intermediate nodes in the hierarchy when using what is believed to be a fast construction strategy (Section 4.2.1).

The efficiency of this bounding volume hierarchy as indicated by Figures 6.6 and 6.5, looks close to the $\mathcal{O}(n \log n)$ when decomposing meshes with a relatively large amount of cells, between which an interface Γ is a simple rectangular polygon divided into segments.

All usage and tests on two dimensional meshes in this thesis have been done using a bounding volume hierarchy designed for three dimensional entities. This limits the relevance of the efficiency tests, but *emphasizes* the modularity and practicality of this approach. Furthermore, only a few meshes typical to the finite element method has been tested, and particularly only meshes typical to the overlapping meshes case. Functionality for the interesting case of 1D networks of intervals submerged in a background mesh (Chapter 3), although probably easy to implement in a bounding volume hierarchy, has not been developed, thoroughly presented or benchmarked.

7.1.2 Implementational considerations when detecting intersections

From a practical perspective, this thesis argues to use the implementation of the bounding volume hierarchy for intersection detection on meshes typical to the finite element method – this at least compared to the implementation of the advancing front algorithm also presented and benchmarked. The advancing front algorithms are here deemed to be cumbersome and lacking modularity, both conceptually and in code, for detecting intersections as it is presented here.

An implementation of an advancing front algorithms for meshes of different dimensionality seems far from trivial to implement, and warrant standalone *closed-loop* implementation for most such combinations. A bounding volume hierarchy easily covers many of these cases once you have just one implementation. The author cannot envision algorithms of complexity $\mathcal{O}(n)$ with the same implementational practicality and modularity as a bounding volume hierarchy or other partitioning scheme in this regard, since any $\mathcal{O}(n)$ algorithm will most likely rely on exploiting locality information encoded in mesh connectivity and thus be bound to the particular representation of a mesh.

The implemented algorithms are here not presented with benchmarks comparing them to implementations by other programmers. This is as intended as not to make this thesis a speed competition. Having said that, comparisons have been made throughout the development – and this bounding volume hierarchy performs *well* in comparisons to others – some are faster, some are slower, naturally also depending on the geometric input. No attempt has been made to properly optimize the data structures employed in these algorithms to be *cache friendly* [22, 74], and this impedes in particularly the speed of the construction algorithm, Algorithm 3.

No thorough investigation of numerical robustness has been performed in this thesis, although some robustness concerns have been expressed in Section 4.2.4: *Detecting the intersection of primitives.* Some functionality increasing numerical robustness have been proposed, but it has not been tested nor has it been implemented to any considerable extent.

7.1.3 Mesh decomposition

This thesis has shown that a greedy triangulation algorithm such as Algorithm 9 (Section 4.6.2) is able to return a triangulation of cut entities in a satisfactory manner for the overlapping meshes case, with the assumption that the mesh sizes are compatible over the interface (Section 3.4). Whether or not the amount of entities returned from such an algorithm is problematic if it is generalized to 3D is still an unanswered question.

7.2 The Nitsche method

Despite the fact that a fairly general geometrical framework capable of handling many unfitted geometries has been developed here, the framework has only been applied to the Nitsche method on overlapping meshes. Furthermore, this thesis has been brief in its treatment of the formulation of the Nitsche method as this theory is here primarily used as a backdrop for developing geometry algorithms. It is thus skipping a presentation of often used stabilization methods and important techniques to improve the convergence of this method [40, 12, 39].

The code presented here is part of a much larger framework for doing numerical analysis, and much of the functionality developed within the FEniCS project needed for these flexible geometrical approaches have been implemented with brand new codes over the course of writing this thesis. The library LibCutFEM recently developed by Massing and others [14] have been given a good run in conjunction with the fairly rudimentary implementational quality of the geometry decomposition algorithms developed here. These development processes have for the most part been successful, an indication of which is the convergence and patch tests in Sections 6.1.4 and 6.1.5.

More attached to finite element analysis than to computational geometry, this thesis has shown that integrating over a messy triangulation returned from a greedy triangulation algorithm such as Algorithm 9 is a viable strategy for cut entities. The algorithms already mentioned potential pitfall of returning a triangulation with an excessive amount of triangles, is not seen to be problematic in the cases of integration presented here, nor is the shape of the triangles it returns.

7.3 Further work

Based on the measured performance of different bounding volumes – at different location in the bounding volume hierarchy – further work is proposed on a *hybrid* bounding volume hierarchy. A hybrid bounding volume hierarchy using the *embedded* 6DOP in a 26DOP as bounding volumes at intermediate nodes in the hierarchy and the whole 26DOP at leaf nodes is seen as a plausible *best of both worlds* approach to further development. A single entity, such as a hybrid volume containing a simple bounding volume like a 6DOP that will be accessed the most, is seen to be a practical entity to optimize and perhaps attempt to parallelize code for. A selection of bounding volumes, as is available in the implementation of the bounding volume hierarchy presented here, is thus not something deemed necessary in a robust and easily generalized future implementation. A rudimentary precursor to this further development has been performed by augmenting the AABB hierarchy currently presented in [54] by 26DOP predicates at the leaf node level. This has been shown to produce the same kind of averaging as observed in Section 4.5.5, increasing both efficiency and reducing worst-case behavior as described above. This is interesting since the this current FEniCS version of this hierarchy is already quite optimized.

A rewrite would furthermore elicit a more comprehensive templated implementational approach, where bounding volume constructors, collision tests and instructions on how to build bounding volumes around arbitrary geometrical entities can be injected in to a small and general hierarchy. Containing this in a class almost without any dependencies is possible. A small bounding volume hierarchy like this, itself completely oblivious of the primitives it contains, but optimized towards the typical role of meshes in finite element analysis, is seen as the natural way forward for extended usability and modularity. This can be said to already be the general idea of this technique, but is something that is often overlooked in actual implementations, and certainly overlooked in the implementation made during the course of development in this thesis. A small and portable implementation like this, would immediately be beneficial to interesting research such as that of Cattaneo and Zunino [9], both in terms of reducing time spent developing algorithms and reducing time spent computing stuff.

More work is further proposed on the triangulation algorithm, Algorithm 9, irrespective of the fact that it seems sufficient for integration in the cases presented here. By visually inspecting Figure 4.17 of the returned triangulation it can be seen that many triangles in this triangulation together form larger triangles. Reducing the number of triangles that is returned by this routine should therefore be possible via some kind intermediate pruning routine, either at the very end of the routine, or at each step after a triangulation conforming to a single facet segment has been made.

Hopefully these codes and descriptions will make solving partial differential equations on unfitted geometries a little more accessible to everyone – like the Nitsche method already does.

Appendix A

Source code

A.1 Poisson's equation with weakly enforced boundary conditions

main.cpp - Main program.

```
#include <dolfin.h>
#include "PoissonNitsche.h"

using namespace dolfin;

// Source term (right-hand side)
class Source : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    float k_0 = 1;
    float k_1 = 1;
    values[0] = DOLFIN_PI*DOLFIN_PI*(k_0*k_0*sin((DOLFIN_PI)*x[0])
                *sin(DOLFIN_PI*x[1])
                + k_1*k_1*sin(DOLFIN_PI*x[0])*sin(DOLFIN_PI*x[1]));
  }
};

int main()
{
  // Create mesh and function space
  UnitSquareMesh mesh(10, 10);
  Poisson::FunctionSpace V(mesh);

  // Define boundary condition
  Constant u0(0.0);

  // Define variational forms
  Poisson::BilinearForm a(V, V);
  Poisson::LinearForm L(V);

  Source f;
  L.f = f;
  Constant gamma(1.0);
  Constant g(0);
  L.g = g;
  L.gamma = gamma;
  a.gamma = gamma;

  // Compute solution
  Function u(V);
  solve(a == L, u);

  // Plot solution
  plot(u);
  interactive();

  return 0;
}
```

PoissonNitsche.ufl - Unified Form Language file that generates PoissonNitsche.h

```
# Compile this form with FFC: ffc -l dolfin PoissonNitsche.ufl.

element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)

f = Coefficient(element)
g = Coefficient(element)
h = Coefficient(element)
n = element.cell().n
h = 2.0*Circumradius(triangle)

gamma = Coefficient(element)

a = (inner(grad(u), grad(v))*dx
    - inner(dot(grad(u),n), v)*ds
    - inner(dot(grad(v),n), u)*ds
    + gamma/h * inner(u,v)*ds)
L = (inner(f,v)*dx
    - inner(dot(grad(v),n), g)*ds
    + gamma/h * inner(g,v)*ds)
```

A.2 Triangulation algorithm verification

Verification code for volume (area) of meshed domain.

```
#include <dolfin.h>
#include <cutfem.h>
void testVolume()
{
    // Original meshes
    double volume_mesh1;
    for (dolfin::CellIterator cellit(*overlapping_meshes->mesh(0));
         !cellit.end(); ++cellit)
    {
        dolfin::Cell cell(*overlapping_meshes->mesh(0),
                          cellit->global_index());

        volume_mesh1 += cell.volume();
    }
    double volume_mesh2;
    for (dolfin::CellIterator cellit(*overlapping_meshes->mesh(1));
         !cellit.end(); ++cellit)
    {
        dolfin::Cell cell(*overlapping_meshes->mesh(1),
                           cellit->global_index());

        volume_mesh2 += cell.volume();
    }

    // T_0
    double volume_t_0;
    for (dolfin::CellIterator cellit(*overlapping_meshes->mesh(0));
         !cellit.end(); ++cellit)
    {
        // If cell_marker = 0, it is in T_0.
        if(overlapping_meshes->
                   cell_marker(0).get()[0][cellit->global_index()] == 0)
        {
            dolfin::Cell cell(*overlapping_meshes->mesh(0),
                              cellit->global_index());
            volume_t_0 += cell.volume();
        }
    }

    // T_1_CUT
    double volume_local_triangulation;
    if(overlapping_meshes->cut_mesh_and_parent_mesh_ids(0).first->size(2) > 1)
    {
        for (dolfin::CellIterator cellit(*overlapping_meshes->
              cut_mesh_and_parent_mesh_ids(0).first); !cellit.end(); ++cellit)
        {
            dolfin::Cell cell(*overlapping_meshes->
              cut_mesh_and_parent_mesh_ids(0).first,
                                    cellit->global_index());
            dolfin::VertexIterator v(cell);
            volume_local_triangulation += cell.volume();
        }
    }

    double sum_untriangulated = volume_mesh1 - volume_mesh2;
    cout << "Sum: mesh - collidingMesh" << endl;
    std::cout << std::setprecision (30) << sum_untriangulated << std::endl;

    cout << "Sum: submesh T_0 + local triangulation" << endl;
    double sum_triangulated = volume_t_0 + volume_local_triangulation;
    std::cout << std::setprecision (30) << sum_triangulated << std::endl;

    double sum_difference = sum_untriangulated - sum_triangulated;
    cout << "Sum: difference" << endl;
    cout << std::setprecision (30)<< std::fixed << sum_difference << endl;

    double precision = 5.96e-14;
    cout << "Float precision" << endl;
    cout << std::setprecision (30)<< std::fixed << precision << endl;
}
```

square_ellipse.py - Generation code for ellipse and square mesh.

```
from dolfin import *

from numpy import linspace
a = 0.25
b = 0.35
edge_points = []
center = Point(0.5,0.5)
angle = pi/2.0 + pi/4.0
for theta in linspace(0,2*pi - 0.1,36):
    x = a*cos(theta)*cos(angle) - b*sin(theta)*sin(angle)
    y = a*cos(theta)*sin(angle) + b*sin(theta)*cos(angle)
    edge_points.append(Point(x,y) + center)

square = Rectangle(0., 0., 1., 1.)
ellipse = Polygon(edge_points)

mesh_square = Mesh(square, 10)
mesh_ellipse = Mesh(ellipse, 3)
```

A.3 Poisson's equation on overlapping meshes

main.cpp - Main program.

```
int main()
{
#include <dolfin.h>
#include <cutfem.h>

// Header files generated by ffc
#include "PoissonOLM0.h"
#include "PoissonOLM1.h"
#include "PoissonOLMInterface.h"

// Loading meshes
mesh = new dolfin::Mesh("mesh.xdmf");
collidingMesh = new dolfin::Mesh("mesh2.xdmf");

mesh->init();
collidingMesh->init();

// Creating cutfem::CompositeMesh mesh, of type TriangulatedOverlappingMeshes
// supplying triangulation routine for contained CutMesh entities.
std::shared_ptr<cutfem::CompositeMesh>
              overlapping_meshes(new cutfem::TriangulatedOverlappingMeshes);

std::shared_ptr<dolfin::Mesh>
              ptr_mesh(new dolfin::Mesh(*mesh));
std::shared_ptr<dolfin::Mesh>
              ptr_collidingMesh(new dolfin::Mesh(*collidingMesh));

overlapping_meshes->add(ptr_mesh);
overlapping_meshes->add(ptr_collidingMesh);

// Computing collisions and calculating triangulation.
overlapping_meshes->compute_intersections();

dolfin::Matrix A_2;

// Create function spaces on each mesh and composite function space
PoissonOLM0::FunctionSpace V0(overlapping_meshes->mesh(0));
PoissonOLM1::FunctionSpace V1(overlapping_meshes->mesh(1));

cutfem::CompositeFunctionSpace V_c(overlapping_meshes);
V_c.add(V0);
V_c.add(V1);
V_c.build();

V_c.view(0);
```

```
// Create empty CompositeForm
cutfem::CompositeForm a_c(V_c, V_c);

// Create standard bilinear forms on mesh 0
PoissonOLM0::BilinearForm a0(V0, V0);
dolfin::Constant gamma(gamma_u);
a0.gamma = gamma;


// Get domain marker for mesh 0;
auto cell_marker_0 = overlapping_meshes->cell_marker(0);

// Attach data to standard forms
a0.set_cell_domains(cell_marker_0);

// Compute constrained dofs for V0
auto & constrained_dofs = V_c.constrained_dofs();

// (Candidates are those ones which lives in elements marked with 2).
cutfem::CutFEMTools::compute_constrained_dofs(constrained_dofs,
       *V_c.view(0),
       cell_marker_0.get(),
       2);

std::size_t dof_counter = 0;
for (std::size_t i = 0; i < constrained_dofs.size(); ++i)
   if (constrained_dofs[i])
   {
      ++dof_counter;
   }

cutfem::CompositeFunction vis_constr_dofs(V_c);
cutfem::CutFEMTools::visualise_constrained_dofs(vis_constr_dofs,
       constrained_dofs);

// Add them as CutForm to CompositeForm
a_c.add(std::shared_ptr<cutfem::CutForm>(new cutfem::CutForm(a0)));

// Get the cut mesh describing cut elements in mesh 0
auto cut_mesh_and_parent_ids =
   overlapping_meshes->cut_mesh_and_parent_mesh_ids(0);
auto cut_mesh_0 = cut_mesh_and_parent_ids.first;
auto parent_mesh_ids_0 = cut_mesh_and_parent_ids.second;

dolfin::info("parent_mesh_ids_0 = %d", parent_mesh_ids_0.size());
// Create quadrature rule for mesh 0
std::size_t order = 1;
std::shared_ptr<cutfem::Quadrature> quadrature_a0_domain_1(
    new cutfem::Quadrature(cut_mesh_0->type().cell_type(),
                           cut_mesh_0->geometry().dim(), order));

// Remember that dxq in PoissonOLM0 has domain_id 1!
a_c.cut_form(0)->set_quadrature(1, quadrature_a0_domain_1);
a_c.cut_form(0)->set_cut_mesh(1, cut_mesh_0);
a_c.cut_form(0)->set_parent_mesh_ids(1, parent_mesh_ids_0);

// Create standard bilinear forms on mesh 1
PoissonOLM1::BilinearForm a1(V1, V1);

// Add them as CutForm to CompositeForm
a_c.add(std::shared_ptr<cutfem::CutForm>(new cutfem::CutForm(a1)));

//Add interface form
PoissonOLMInterface::BilinearForm ai(V0, V0);
ai.gamma = gamma;

a_c.add(std::shared_ptr<cutfem::CutForm>(new cutfem::CutForm(ai)));

// Get interface cut mesh and related information, stored as the second
// cut_mesh_and_parent_ids component in TomOverlappingMeshes
auto interface_mesh_and_parent_mesh_ids =
    overlapping_meshes->cut_mesh_and_parent_mesh_ids(1);
auto interface_mesh = interface_mesh_and_parent_mesh_ids.first;
auto interface_parent_mesh_ids = interface_mesh_and_parent_mesh_ids.second;

order = 2;
std::shared_ptr<cutfem::Quadrature> quadrature_interface(
    new cutfem::Quadrature(interface_mesh->type().cell_type(),
                           interface_mesh->geometry().dim(), order));

// Build normal field for quadrature points
std::shared_ptr<cutfem::FacetNormals> interface_facet_normals(
    new cutfem::FacetNormals(interface_mesh->facet_normals(),
                             interface_mesh->geometry().dim(),
                             quadrature_interface->size()));

a_c.cut_form(2)->set_quadrature(0, quadrature_interface);
a_c.cut_form(2)->set_cut_mesh(0, interface_mesh);
a_c.cut_form(2)->set_parent_mesh_ids(0, interface_parent_mesh_ids);
a_c.cut_form(2)->set_facet_normals(0, interface_facet_normals);


// Assemble composite form and compare.
cutfem::CompositeFormAssembler assembler;
assembler.assemble(A_2, a_c);

// Temporary assemble of rhs and solution.
```

```
cutfem::CompositeForm L_c(V_c);

//dolfin::Constant f(1);

// Source term (right-hand side)
class Source : public dolfin::Expression
{
   void eval(dolfin::Array<double>& values,
            const dolfin::Array<double>& x) const
   {
      float k_0 = 1;
      float k_1 = 1;
      values[0] = DOLFIN_PI*DOLFIN_PI
                 *(k_0*k_0*sin((DOLFIN_PI)*x[0])*sin(DOLFIN_PI*x[1])
                 + k_1*k_1*sin(DOLFIN_PI*x[0])*sin(DOLFIN_PI*x[1]));
   }
};

class DirichletBC : public dolfin::Expression
{
   void eval(dolfin::Array<double>& values,
            const dolfin::Array<double>& x) const
   {
      values[0] = 0;
   }
};

PoissonOLM0::LinearForm L0(V0);

// Attach data to standard forms
Source f;
L0.f = f;

DirichletBC g;
L0.g = g;

L0.gamma = gamma;

L0.set_cell_domains(cell_marker_0);
//L0.f = f;

// Add them as CutForm to CompositeForm
L_c.add(std::shared_ptr<cutfem::CutForm>(new cutfem::CutForm(L0)));

// Get the cut mesh describing cut elements in mesh 0

// Create quadrature rule for mesh 0
order = 2;
std::shared_ptr<cutfem::Quadrature> quadrature_L0_domain_1(
   new cutfem::Quadrature(cut_mesh_0->type().cell_type(),
                     cut_mesh_0->geometry().dim(), order));

// Remember that dxq in PoissonOLM0 has domain_id 1!
L_c.cut_form(0)->set_quadrature(1, quadrature_L0_domain_1);
L_c.cut_form(0)->set_cut_mesh(1, cut_mesh_0);
L_c.cut_form(0)->set_parent_mesh_ids(1, parent_mesh_ids_0);

// Create standard bilinear forms on mesh 1
PoissonOLM1::LinearForm L1(V1);
// Reusing source function on overlapping mesh
L1.f = f;

// Add them as CutForm to CompositeForm
L_c.add(std::shared_ptr<cutfem::CutForm>(new cutfem::CutForm(L1)));

dolfin::Vector b_2;
assembler.assemble(b_2, L_c);

cutfem::CompositeFunction u(V_c);

dolfin::solve(A_2, *u.vector(), b_2);
}
```

PoissonOLM0.ufl - UFL program that generates PoissonOLM0.h.

```
# compile with ffc -l dolfin PoissonOLM0.ufl

cell = triangle

V = FiniteElement("CG", cell, 1)

u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)

n = FacetNormal(cell)
h = 2.0*Circumradius(cell)

dxq = dc(1, metadata={"num_cells": 1})

gamma = Coefficient(V)
g = Coefficient(V)

# Standard assembly over uncut cells
a = inner(grad(u), grad(v))*dx(0)
# Nitsche term to enforce weak boundary conditions
# on the (matching) boundary.
a += -inner(dot(grad(u),n), v)*ds
```

```
a += -inner(dot(grad(v),n), u)*ds
a += gamma/h*inner(u,v)*ds

# Integrate same form over cut cells (marked with 1)
a += inner(grad(u), grad(v))*dxq

L = f*v*dx(0) + f*v*dxq - inner(dot(grad(v),n), g)*ds + gamma/h * inner(g,v)*ds
```

PoissonOLMInterface.ufl - UFL program that generates PoissonOLMInterface.h.

```
# compile with ffc -l dolfin PoissonOLMInterface.ufl

cell = triangle

n = FacetNormal(cell)
h = 2.0*Circumradius(cell)

V = FiniteElement("CG", cell, 1)

u = TrialFunction(V)
v = TestFunction(V)

dSq = dc(0, metadata={"num_cells": 2})

gamma = Coefficient(V)

# Averaged version
a = -inner(avg(grad(u)), jump(v, n))*dSq
a += -inner(avg(grad(v)), jump(u, n))*dSq

# One-side flux (better for convergence)
a = -inner(grad(u)("-"), jump(v, n))*dSq
a += -inner(grad(v)("-"), jump(u, n))*dSq
a += gamma/h("-")*inner(jump(u),jump(v))*dSq
```

PoissonOLM1.ufl - UFL program that generates PoissonOLM1.h.

```
# compile with ffc -l dolfin PoissonOLM1.ufl

cell = triangle

V = FiniteElement("CG", cell, 1)

u = TrialFunction(V)
v = TestFunction(V)

# Standard assembly over uncut cells
a = inner(grad(u), grad(v))*dx
```

A.4 Generation code for mesh with large angles

The code for the implementation that follows can be found at:
`https://github.com/tomana/compgeom_olm/tree/master/utils/large_angles`

large__angles.py - Python program generating mesh with large angles.

```python
from dolfin import *
import numpy

mesh = Mesh()
# Using dolfins dynamic mesh editor
editor = DynamicMeshEditor()
editor.open(mesh, 2, 2, 2)
num_cells_x = 8;
num_cells_y = 40;

# Switches used to form lattice in x y space.
switchx = False;
switchy = False;

vert_list = []

# Adding vertices, not necassarily distinct,
# made distinct and put into cells further below.
for j in range(0,num_cells_y):
    switchy = not switchy;
    for i in range(0,num_cells_x):
        # Add vertices
        switchx = not switchx;
        if(switchy):
            switchx = not switchx
        spacer_x = 1.0/float(num_cells_x)
        spacer_y = 1.0/float(num_cells_y)
        if(switchx):
            if(i == 0):
                vert_list.append(((i+1)*spacer_x, j*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+2)*spacer_x, j*spacer_y))
            elif(i == num_cells_x - 1):
                vert_list.append((i*spacer_x, j*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+1)*spacer_x, j*spacer_y))
            else:
                vert_list.append((i*spacer_x, j*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+2)*spacer_x, j*spacer_y))
        else:
            if(i == 0):
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
                vert_list.append(((i+2)*spacer_x, (j+1)*spacer_y))
            elif(i == num_cells_x - 1):
                vert_list.append(((i)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
            else:
                vert_list.append(((i)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
                vert_list.append(((i+2)*spacer_x, (j+1)*spacer_y))

        if(switchy):
            switchx = not switchx

# Sorting so that vertices are unique
set(vert_list)
new_list = list(set(vert_list))

print new_list

cell = 0;
for points in new_list:
    editor.add_vertex(cell, points[0], points[1])
    cell += 1

switchx = False;
switchy = False;

# Now making cells
for j in range(0,num_cells_y):
    switchy = not switchy;
    for i in range(0,num_cells_x):
        vert_list = []
        switchx = not switchx;
        if(switchy):
            switchx = not switchx
        spacer_x = 1.0/float(num_cells_x)
        spacer_y = 1.0/float(num_cells_y)
        if(switchx):
            if(i == 0):
                vert_list.append(((i+1)*spacer_x, j*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+2)*spacer_x, j*spacer_y))
            elif(i == num_cells_x - 1):
                vert_list.append((i*spacer_x, j*spacer_y))
                vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
                vert_list.append(((i+1)*spacer_x, j*spacer_y))
            else:
                vert_list.append((i*spacer_x, j*spacer_y))
```

```
              vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
              vert_list.append(((i+2)*spacer_x, j*spacer_y))
       else:
          if(i == 0):
              vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
              vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
              vert_list.append(((i+2)*spacer_x, (j+1)*spacer_y))
          elif(i == num_cells_x - 1):
              vert_list.append(((i)*spacer_x, (j+1)*spacer_y))
              vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
              vert_list.append(((i+1)*spacer_x, (j+1)*spacer_y))
          else:
              vert_list.append(((i)*spacer_x, (j+1)*spacer_y))
              vert_list.append(((i+1)*spacer_x, (j)*spacer_y))
              vert_list.append(((i+2)*spacer_x, (j+1)*spacer_y))

       if(switchy):
          switchx = not switchx
# Add cell
       editor.add_cell(j*num_cells_x + i, \
                     new_list.index(vert_list[0]) \
                     , new_list.index(vert_list[1]) \
                     , new_list.index(vert_list[2]))

# Close editor
editor.close()

mesh.order()
mesh.init()

file = File("mesh.xml")
file << mesh

plot(mesh)

# Solving a Poisson's equation on this.
V = FunctionSpace(mesh, "Lagrange", 1)

spacer_x = 1.0/float(num_cells_x)
spacer_y = 1.0/float(num_cells_y)

# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS + spacer_x \
      or x[0] > 1.0 - DOLFIN_EPS \
      or x[1] < DOLFIN_EPS \
      or x[1] > 1.0 - DOLFIN_EPS

# Define boundary condition
```

```
u0 = Constant(0.0)
bc = DirichletBC(V, u0, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("10*exp(-(pow(x[0] - 0.5, 2)
              + pow(x[1] - 0.5, 2)) / 0.02)")
g = Expression("sin(5*x[0])")
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution
plot(u)

interactive()
```

A.5 Source code for drawing trees

Source uses openFrameworks [55] toolkit. Put together using strings returned from prints while traversing trees captured in a big string. Using GL2PS [75] to turn OpenGL graphics into PostScript and SVG.:
`https://github.com/tomana/compgeom_olm/tree/master/utils/draw_tree`

A.6 Source code for drawing bounding volumes on mesh

Source uses openFrameworks [55] tookit. Put together using prints returned from a tree captured in a big string. Using GL2PS [75] to turn OpenGL graphics into PostScript and SVG.:
`https://github.com/tomana/compgeom_olm/tree/master/utils/draw_volumes_on_mesh`

A.7 Generation code for structured mesh

Function for generating structured meshes of square with rotated mesh in the middle.

```
void new_structuredmeshes(size_t n)
{
   size_t N = n;
   // Build background mesh
```

```
    double a = 3.0;
    double b = -3.0;
    mesh->clear();
    collidingMesh->clear();
    std::shared_ptr<dolfin::Mesh> mesh0(
        new dolfin::RectangleMesh(a, a, b, b, 3 * N, 3 * N));
    mesh0->init(mesh0->topology().dim() - 1, mesh0->topology().dim());

    // Create sub mesh
    std::shared_ptr<dolfin::CellFunction<std::size_t> > domain_marker_0
    (new dolfin::CellFunction<std::size_t>(*mesh0, 0));

    InnerDomain().mark(*domain_marker_0, 2);
    std::shared_ptr<dolfin::SubMesh> mesh1(
        new dolfin::SubMesh(*mesh0, *domain_marker_0, 2));

    // Scale overlapping mesh
    double scalefactor = 1.5;
    dolfin::MeshGeometry& geometry = mesh1->geometry();
    for (std::size_t i = 0; i < geometry.size(); i++)
    {
        // Get coordinate
        double* x = geometry.x(i);
        // Scale
        const double x0 = x[0]*scalefactor;
        const double x1 = x[1]*scalefactor;
        // Store coordinate
        x[0] = x0;
        x[1] = x1;
    }
    // Copy meshes
    mesh = new dolfin::Mesh(*mesh0);
    collidingMesh = new dolfin::Mesh(*mesh1);
    // Rotate overlapping mesh
    collidingMesh->rotate(40,2);
    mesh->init();
    collidingMesh->init();
}
```

Bibliography

[1] Paul T Boggs, Alan Althsuler, Alex R Larzelere, Edward J Walsh, Ruuobert L Clay, and Michael F Hardwick. DART system analysis. 2005.

[2] J Austin Cottrell, Thomas JR Hughes, and Yuri Bazilevs. *Isogeometric analysis: toward integration of CAD and FEA*. John Wiley & Sons, 2009.

[3] Erik Burman and Peter Hansbo. Fictitious domain finite element methods using cut elements: I. A stabilized Lagrange multiplier method. *Computer Methods in Applied Mechanics and Engineering*, 199(41):2680–2686, 2010.

[4] M Tur, J Albelda, E Nadal, and JJ Ródenas. Imposing Dirichlet boundary conditions in hierarchical Cartesian meshes by means of stabilized Lagrange multipliers. *International Journal for Numerical Methods in Engineering*, 2014.

[5] Patrick E Farrell. *Galerkin projection of discrete fields via supermesh construction*. PhD thesis, Imperial College London, 2009.

[6] Ramsharan Rangarajan, Adrián Lew, and Gustavo C Buscaglia. A discontinuous-Galerkin-based immersed boundary method with non-homogeneous boundary conditions and its application to elasticity. *Computer Methods in Applied Mechanics and Engineering*, 198(17):1513–1534, 2009.

[7] August Johansson and Mats G Larson. A high order discontinuous Galerkin Nitsche method for elliptic problems with fictitious boundary. *Numerische Mathematik*, 123(4):607–628, 2013.

[8] Adrián J Lew and Gustavo C Buscaglia. A discontinuous-Galerkin-based immersed boundary method. *International Journal for Numerical Methods in Engineering*, 76(4):427–454, 2008.

[9] Laura Cattaneo and Paolo Zunino. Computational models for fluid exchange between microcirculation and tissue interstitium. *Networks and Heterogeneous Media*, 2013.

[10] Charles S Peskin. The immersed boundary method. *Acta numerica*, 11, 2002.

[11] J Nitsche. Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 36, pages 9–15. Springer, 1971.

[12] A. Massing, M.G. Larson, A. Logg, and M.E. Rognes. A stabilized Nitsche overlapping mesh method for the Stokes problem. *Num. Math.*, page 1–29, 2014.

[13] A. Massing, M.G. Larson, and A. Logg. Efficient implementation of finite element methods on non-matching and overlapping meshes in 3D. *SIAM J. Sci. Comput.*, 35(1):C23–C47, 2013.

[14] E. Burman, S. Claus, P. Hansbo, M.G. Larson, and A. Massing. CutFEM: discretizing geometry and partial differential equations. *Submitted to Int. J. Numer. Meth. Engng*, 2014.

[15] Anita Hansbo, Peter Hansbo, and Mats G Larson. A finite element method on composite grids based on Nitsche's method. *ESAIM: Mathematical Modelling and Numerical Analysis*, 37(03):495–514, 2003.

[16] Peter Hansbo. Nitsche's method for interface problems in computational mechanics. *GAMM-Mitteilungen*, 28(2):183–206, 2005.

[17] Mika Juntunen and Rolf Stenberg. Nitsche's method for general boundary conditions. *Mathematics of computation*, 78(267):1353–1374, 2009.

[18] CGAL: Computational Geometry Algorithms Library. *http://www.cgal.org.*

[19] Fabio Ganovelli, Federico Ponchio, and Claudio Rocchini. Fast tetrahedron-tetrahedron overlap algorithm. *Journal of Graphics Tools*, 7(2):17–25, 2002.

[20] James T Klosowski, Martin Held, Joseph SB Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *Visualization and Computer Graphics, IEEE Transactions on*, 4(1):21–36, 1998.

[21] Martin J Gander and Caroline Japhet. An algorithm for non-matching grid projections with linear complexity. In *Domain Decomposition Methods in Science and Engineering XVIII*, pages 185–192. Springer, 2009.

[22] Christer Ericson. *Real-time collision detection*. Taylor & Francis US, 2005.

[23] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM Siggraph Computer Graphics*, volume 21, pages 163–169. ACM, 1987.

[24] Susanne C Brenner and Larkin Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer, 2008.

[25] M.G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications.* Texts in Computational Science and Engineering. Springer, 2013.

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

[27] Isaac Fried. Condition of finite element matrices generated from nonuniform meshes. *Aiaa Journal*, 10(2):219–221, 1972.

[28] Pascal J Frey, Houman Borouchaki, and Paul Louis George. Delaunay tetrahedralization using an advancing-front approach. In *5th International Meshing Roundtable*, pages 31–48. Citeseer, 1996.

[29] Michael Murphy, David M Mount, and Carl W Gable. A point-placement strategy for conforming Delaunay tetrahedralization. *International Journal of Computational Geometry & Applications*, 11(06):669–682, 2001.

[30] Hang Si. TetGen - A quality tetrahedral mesh generator and three-dimensional delaunay triangulator. 2006.

[31] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[32] Joachim Schöberl. NETGEN: An advancing front 2D/3D-mesh generator based on abstract rules. *Computing and visualization in science*, 1(1):41–52, 1997.

[33] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied computational geometry towards geometric engineering*, pages 203–222. Springer, 1996.

[34] L Antiga and DA Steinman. VMTK: Vascular Modeling Toolkit. *http://www.vmtk.org*.

[35] pyFormex - program for generating, transforming and manipulating large geometrical models of 3D structures by sequences of mathematical operations. *http://www.nongnu.org/pyformex*. Accessed: 2014-06-14.

[36] A. Massing, M.G. Larson, A. Logg, and M.E. Rognes. An overlapping mesh finite element method for a fluid-structure interaction problem. *Submitted to CAMCoS, available as arXiv preprint arXiv:1311.2431*, 2013.

[37] Sven Groß and Arnold Reusken. An extended pressure finite element space for two-phase incompressible flows with surface tension. *Journal of Computational Physics*, 224(1):40–58, 2007.

[38] Luca Antiga, Joaquim Peiró, and David A Steinman. From image data to computational domains. In *Cardiovascular Mathematics*, pages 123–175. Springer, 2009.

[39] Erik Burman and Peter Hansbo. Fictitious domain finite element methods using cut elements: II. A stabilized Nitsche method. *Applied Numerical Mathematics*, 62(4):328–341, 2012.

[40] Jarle Sogn. Stabilized finite element methods for the brinkman equation on fitted and ficititious domains. 2014.

[41] Chris Lomont. Fast inverse square root (found in the Quake 3 engine.). *Tech-315 nical Report*, 2003.

[42] Blender free and open source 3D animation suite. *http://www.blender.org/*.

[43] Thomas Larsson and Tomas Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, 30(3):450–459, 2006.

[44] Stefan Gottschalk. *Collision queries using oriented bounding boxes.* PhD thesis, The University of North Carolina, 2000.

[45] Michael S Paterson and F Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry*, 5(1):485–503, 1990.

[46] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[47] Hank Weghorst, Gary Hooper, and Donald P Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics (TOG)*, 3(1):52–69, 1984.

[48] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13. ACM, 2009.

[49] Eric Larsen, Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Fast distance queries with rectangular swept sphere volumes. In *Robotics and Automation, 2000. Proceedings. ICRA'00.*, volume 4, pages 3719–3726. IEEE, 2000.

[50] Jack Ritter. An efficient bounding sphere. In *Graphics gems*, pages 301–303. Academic Press Professional, Inc., 1990.

[51] Tomas Möller. A fast triangle-triangle intersection test. *Journal of graphics tools*, 2(2):25–30, 1997.

[52] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. In *Algorithms–ESA 2004*, pages 702–713. Springer, 2004.

[53] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method.* Springer, 2012.

[54] Anders Logg, Garth N. Wells, and Johan Hake. *DOLFIN: a C++/Python Finite Element Library*, chapter 10. Springer, 2012.

[55] openFrameworks open source C++ toolkit for creative coding. *http://www.openframeworks.cc.*

[56] OpenGL (Open Graphics Library). *http://www.opengl.org/.*

[57] GNU Standard C++ Library v3. *http://gcc.gnu.org/libstdc++/.*

[58] MeshKit 1.0 - Open-source library of mesh generation functionality. *https://trac.mcs.anl.gov/projects/fathom/wiki/MeshKit.*

[59] Timothy James Tautges, Corey Ernst, Clint Stimpson, Ray J Meyers, and Karl Merkley. MOAB: a mesh-oriented database. Technical report, Sandia National Laboratories, 2004.

[60] Aristides AG Requicha and Herbert B Voelcker. Constructive solid geometry. 1977.

[61] Paul Bourke. Polygonising a scalar field. *http://paulbourke.net/geometry/polygonise/*, 1994.

[62] C++ implementations of computational geometry algorithms related to mesh decomposition for FEM on overlapping meshes. *https://github.com/tomana/compgeom_olm/.*

[63] Y Sudhakar and Wolfgang A Wall. Quadrature schemes for arbitrary convex/concave volumes and integration of weak form in enriched partition of unity methods. *Computer Methods in Applied Mechanics and Engineering*, 2013.

[64] Ted Belytschko, Nicolas Moës, Shuji Usui, and Chandu Parimi. Arbitrary discontinuities in finite elements. *International Journal for Numerical Methods in Engineering*, 50(4):993–1013, 2001.

[65] Natarajan Sukumar and Ted Belytschko. Arbitrary branched and intersecting cracks with the extended finite element method. *Int. J. Numer. Meth. Engng*, 48:1741–1760, 2000.

[66] John Dolbow and T Belytschko. A finite element method for crack growth without remeshing. *Int. J. Numer. Meth. Engng*, 46(1):131–150, 1999.

[67] N Sukumar, N Moës, B Moran, and T Belytschko. Extended finite element method for three-dimensional crack modelling. *International Journal for Numerical Methods in Engineering*, 48(11):1549–1570, 2000.

[68] QZ Xiao and BL Karihaloo. Improving the accuracy of XFEM crack tip fields using higher order quadrature and statically admissible stress recovery. *International Journal for Numerical Methods in Engineering*, 66(9):1378–1410, 2006.

[69] Sergio Zlotnik and Pedro Díez. Hierarchical X-FEM for n-phase flow (n $>$ 2). *Computer Methods in Applied Mechanics and Engineering*, 198(30):2329–2338, 2009.

[70] B Müller, F Kummer, and M Oberlack. Highly accurate surface and volume integration on implicit domains by means of moment-fitting. *International Journal for Numerical Methods in Engineering*, 96(8):512–528, 2013.

[71] Y Sudhakar, JP Moitinho de Almeida, and Wolfgang A Wall. An accurate, robust, and easy-to-implement method for integration over arbitrary polyhedra: Application to embedded interface methods. *Journal of Computational Physics*, 2014.

[72] Susanne Claus. *Numerical simulation of complex viscoelastic flows using discontinuous galerkin spectral/hp element methods.* PhD thesis, Cardiff University, 2013.

[73] Vinh Phu Nguyen, Pierre Kerfriden, Susanne Claus, Stephane Pierre-Alain Bordas, et al. Nitsche's method method for mixed dimensional analysis: conforming and non-conforming continuum-beam and continuum-plate coupling. 2013.

[74] Joon-Sang Park, Michael Penner, and Viktor K Prasanna. Optimizing graph algorithms for improved cache performance. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9):769–782, 2004.

[75] GL2PS: an OpenGL to PostScript printing library. *http://www.geuz.org/gl2ps/.*