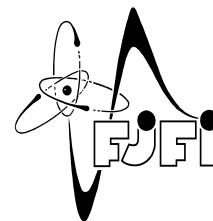




ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská



Numerická simulace vícefázového proudění na nestrukturovaných sítích s libovolnou topologií ve 3D

Numerical Simulation of Multiphase Flow on 3D Unstructured Meshes with an Arbitrary Topology

Diplomová práce

Autor: **Tomáš Jakubec**
Vedoucí práce: **Ing. Pavel Strachota, Ph.D.**
Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Tomáš Jakubec
Studijní program:	Aplikace přírodních věd
Obor:	Matematické inženýrství
Název práce (česky):	Numerická simulace vícefázového proudění na nestrukturovaných sítích s libovolnou topologií ve 3D
Název práce (anglicky):	Numerical simulation of multiphase flow on 3D unstructured meshes with an arbitrary topology

Pokyny pro vypracování:

1. Implementujte knihovnu pro práci s obecnou trojrozměrnou nestrukturovanou sítí v C++, tj. s buňkami ve tvaru libovolného mnohostěnu.
2. Pro ověření správné funkcionality implementace vytvořte unit testy.
3. Implementujte načítání a ukládání geometrie sítě a výpočetních dat v některém ze standardních formátů (např. VTK).
4. Navrhněte schéma konečných objemů na nestrukturovaných 3D sítích pro úlohu vícefázového proudění popisující procesy fluidizace. S použitím tohoto schématu a vhodné časové diskretizace (např. pomocí Rungeovy-Kuttovy metody vyššího řádu) vytvořte příslušný numerický algoritmus.
5. Rozpracujte vhodný způsob masivní paralelizace výpočtů na nestrukturované síti, např. pomocí MPI, OpenCL nebo CUDA.
6. Ověřte funkčnost vyvinutého numerického algoritmu při simulacích dynamiky fluidního lože ve spalovací komoře fluidního kotle.

Doporučená literatura:

1. D. Gidaspow, Multiphase flow and fluidization: Continuum and kinetic theory description. Academic Press, 1994.
2. J. Blazek, Computational Fluid Dynamics: Principles and Applications (3rd ed.). Elsevier, 2015.
3. F. Moukalled, L. Mangani, M. Darwish et al., The finite volume methods in computational fluid dynamics. Springer, 2016.
4. S. Prata, C++ Primer Plus (6th ed.). Pearson Education, Addison Wesley, 2012.
5. G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.
6. J. Cheng, M. Grossman, and T. McKercher. Professional CUDA C Programming. Wrox, 2014.

Jméno a pracoviště vedoucí diplomové práce:

Ing. Pavel Strachota, Ph. D.
FJFI ČVUT v Praze, Trojanova 13, 120 00 Praha 2

Jméno a pracoviště konzultanta:

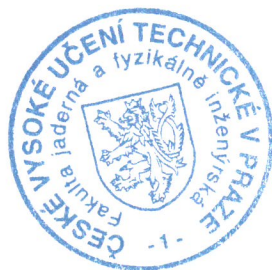
Datum zadání diplomové práce: 31.10.2019

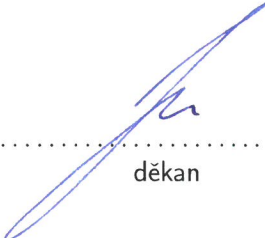
Datum odevzdání diplomové práce: 4.5.2020

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 14. října 2019


.....
garant oboru
.....
vedoucí katedry




.....
děkan

Poděkování:

Chtěl bych zde poděkovat především svému školiteli Ing. Pavlovi Strachotovi, Ph.D. za pečlivost, ochotu, vstřícnost a odborné i lidské zázemí při vedení mé diplomové práce.

Čestné prohlášení:

Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl jsem všechnu použitou literaturu.

V Praze dne 24. června 2020

Tomáš Jakubec

Název práce:

Numerická simulace vícefázového proudění na nestrukturovaných sítích s libovolnou topologií ve 3D

Autor: Bc. Tomáš Jakubec

Obor: Matematické inženýrství

Zaměření: Matematické modelování

Druh práce: Diplomová práce

Vedoucí práce: Ing. Pavel Strachota Ph.D.
Katedra matematiky
Fakulta jaderná a fyzikálně inženýrská
České vysoké učení technické v Praze

Abstrakt: Tato diplomová práce představuje tvorbu numerické C++ knihovny pracující s obecnými nestrukturovanými sítěmi a její následné využití pro simulování dvoufázového proudění. Zprvu je pozornost soustředěna na vytvoření grafového popisu nestrukturované sítě za účelem výběru vhodné datové struktury pro následnou implementaci. Dále se text věnuje detailnímu popisu implementace moderní šablonové C++ knihovny schopné pracovat s obecnými sítěmi, a to dokonce v obecné dimenzi. Knihovna je od počátku koncipována s ohledem na využití v paralelních výpočetních algoritmech včetně výpočtů na grafických kartách. Navíc poskytuje nástroje, které zjednodušují tvorbu numerických algoritmů při zachování výpočetního výkonu. Druhá část práce popisuje aplikaci metody konečných objemů na problém dvoufázového proudění. Nakonec je daný problém numericky řešen ve 2D i 3D s využitím vyvinuté knihovny a několika typů nestrukturovaných sítí.

Klíčová slova: nestrukturovaná síť, C++, šablonové metaprogramování, paralelizace, metoda konečných objemů, dvoufázové proudění

Title:

Numerical Simulation of Multiphase Flow on 3D Unstructured Meshes with an Arbitrary Topology

Author: Bc. Tomáš Jakubec

Abstract: This master thesis presents the development of a C++ numerical library working with general unstructured meshes and its application in the simulation of a two-phase flow. At first, the attention is paid to the creation of a graph formalism representing an unstructured mesh, which is utilized to choose a suitable data structure for the subsequent implementation. Next, the implementation of the C++ library using modern paradigms is described in detail. This library is capable of representing unstructured meshes with general topology and dimension. It is designed for use in parallel computational algorithms including GPGPU computations. In addition, it provides clever tools simplifying the creation of numerical codes without any impact on performance. In its second part, the thesis presents the problem of two-phase flow and its numerical solution by the finite volume method. Finally, the simulations on 2D and 3D meshes by means of the developed library are demonstrated.

Key words: unstructured mesh, C++, template metaprogramming, parallelization, finite volume method, two-phase flow

Contents

Introduction	11
1 Unstructured Mesh Representation	13
1.1 General Unstructured Polyhedral Meshes	15
1.2 Notation on Unstructured Mesh	15
1.3 Data Structure Representing Meshes with Arbitrary Topology	17
1.3.1 Graph Description	17
1.3.2 Possible Representations	19
1.3.3 Representation of Choice	21
2 The GTMesh Library	25
2.1 Project Architecture	25
2.2 The MeshElements Structure	29
2.2.1 Scheme of the MeshElement structure	29
2.2.2 Preparation of Generic MeshElement Properties	30
2.2.3 Definition of the MeshElement Structure	31
2.2.4 Construction of the MeshElements structure	38
2.2.5 Mesh Boundary	39
2.3 Data Associated to the Mesh	41
2.3.1 Properties of the MeshDataContainer Class	42
2.3.2 Construction of MeshDataContainer	42
2.3.3 Generation of MeshDataContainer Using std::integer_sequence	44
2.4 Mesh Algorithms	52
2.4.1 Element Access and the MeshApply Class	52
2.4.2 Mesh Connections and the MeshConnections Class	61
2.4.3 Elements Neighborhood and the MeshNeighborhood Class	62
2.4.4 Mesh Coloring and the ColorMesh Class	68
2.4.5 Element Center Calculation	72
2.4.6 Element Measure Calculation and the computeMeasures Function	77
2.4.7 Elements Orientation and the computeFaceNormals Function	85
2.5 Mesh Import and Export	88
2.5.1 VTK format	88
2.5.2 FPMA format	95
2.5.3 Mesh Signature	97
2.6 UnstructuredMesh as the MeshElements Wrapper	99
2.7 3D Meshes with Non-planar Faces	99
3 Class Traits	103
3.1 The MemberAccess Class	104
3.2 Traits as a Tuple of MemberAccess	106
3.3 Default Traits	110

3.3.1	DefaultIOTraits and DefaultArithmeticTraits	113
3.3.2	Existence of Default Class Traits	114
3.3.3	Macros Creating Default Class Traits	115
4	Class Traits Applications	123
4.1	Debugging System and Automatic Data I/O	123
4.1.1	The VariableExport Class	123
4.1.2	Loggers	125
4.2	Numerical Algorithms Based on Class Traits	131
4.3	The Runge-Kutta-Merson Solver	137
5	Numerical Solution of Compressible Two-phase Flow	139
5.1	Governing Equations	139
5.1.1	Initial Conditions	141
5.1.2	Boundary Conditions	141
5.2	Numerical Scheme	142
5.2.1	Unstructured Mesh Notation	142
5.2.2	Finite Volume Method on Unstructured Meshes	143
5.2.3	Treatment of Non-planar Faces	151
5.2.4	Temporal Discretization	152
5.3	Realization of the Computation in the GTMesh framework	152
6	Parallel Implementation on GPU	159
6.1	Adaptation of UnstructuredMesh to GPU	160
6.2	Adaptation of MeshDataContainer to GPU	160
6.3	Automatic Conversion Between AoS and SoA	161
7	Simulations	165
7.1	Comparison of Gas Flow on Several Meshes	165
7.2	Two-Phase Flow in 2D	170
7.3	Two-Phase Flow in 3D	173
	Conclusion	177
	Bibliography	181
A	Distribution of GTMesh	183

Introduction

Unstructured meshes are indispensable for numerical simulations on domains with complex geometry. Recently, the use of meshes with general topology has become increasingly popular [37, 55, 51]. Several software projects exist that provide multiphysics simulations on unstructured meshes [17, 1, 2]. However, the possibilities of those products may be limited in some specific cases of use. In particular, their support for extremely demanding simulations requiring the use of massively parallel hybrid CPU/GPU environments is very poor. To take advantage of the most powerful compute systems, specialized software packages [19, 14] combined with in-house code are commonly used. Therefore, the aim of this master thesis is to develop a general numerical library for working with unstructured meshes [28, 37, 55] with general topology and dimension. The library utilizes modern C++ paradigms [52, 3, 5, 8, 7] with emphasis on easy use while still maintaining the computational efficiency and generality. The library is developed with respect to the potential support of parallelization (GPU through CUDA framework [15, 34] or MPI [16]). The aim of the library is to provide a general framework to support multiple of numerical methods, e.g., finite element method (FEM) [32] or finite volume method (FVM) [45, 30, 42].

Next, in relation to fluidized bed (FB) boilers [25, 47, 26, 48, 41, 27] a problem of two-phase flow [29, 36] is discussed in this work. The two-phase flow is discretized using FVM and the obtained system of ordinary differential equations is numerically solved by the Merson version of the 4th order Kunge-Kutta solver with an adaptive time step [49, 33, 35]. The obtained scheme is implemented using the prepared library.

Chapter 1 discusses general unstructured meshes. It presents the notation on unstructured meshes and defines the basic qualities of the meshes. It introduces the graph description of an unstructured mesh which is utilized during the design of the data structures utilized in the library.

Chapter 2 presents in detail the implementation of the structure storing an unstructured mesh in the developed library named *GTMesh*. Big effort was made to explain the utilized programming concepts and architecture which were the key part of the success. This chapter also introduces an intuitive system of mapping data to the mesh and the implementation of basic mesh algorithms (e.g., determination of arbitrary connections, neighbors and proper coloring and calculation of the elements centers and measures). Then, the import and export of the unstructured mesh and the mapped data are described. Finally, this chapter introduces the work with the 3D unstructured meshes with non-planar faces.

Chapter 3 describes the system of annotation of C++ structures and classes in terms of providing a standardized interface to the data stored in the attributes of classes and structures.

Chapter 4 presents the applications of the tool developed in the third chapter. The first application is a general logging tool created primarily for the debugging purposes. The second application is to create automatically generated element-wise arithmetic operations for classes and structures having the annotation defined. Finally, the Merson version of the 4th order Runge-Kutta solver with an adaptive time step is presented with emphasis on the possible use on classes with annotation defined.

In Chapter 4, the problem of two-phase flow in a 2D and 3D is presented. Afterward, the discretization by FVM follows. Then, the application of the obtained equations is discussed for

the case of a 3D unstructured mesh with non-planar faces. In the end, the implementation of the problem by using the concepts from GTMesh is commented.

Chapter 6 briefly discusses the possibility of adaptation of GTMesh to GPU with utilization of the TNL project [22]. Despite the brief description of the problem, the fundamental changes needed for the adaptation to GPU are presented in detail.

Chapter 7 presents the results of the performed simulations. At first, the comparison of simulations of gas flow performed on different meshes (structured and unstructured) is presented. Then, simulations of two-phase flow in 2D and 3D computational domains representing the combustion chamber of a FB boiler are demonstrated.

This work was partially supported by the project "Research centre for low-carbon energy technologies" (Reg. No. CZ.02.1.01/0.0/0.0/16-019/0000753) and the project "Centre of Advanced Applied Sciences" (Reg. No. CZ.02.1.01/0.0/0.0/16-019/0000778), provided by the Ministry of Education, Youth and Sports of the Czech Republic under the OP RDE program and co-funded by the European Union.

Chapter 1

Unstructured Mesh Representation

In numerical mathematics, the meshes are used to tessellate a computational domain. We will denote the computational domain as $\Omega \subset \mathbb{R}^d$. Generally the mesh consists of a set of elements (entities) as cells, faces, edges and vertices. In the mesh, those elements are topologically connected according to the position. These connections determine the constructions of elements. See an example in Figure 1.6.

In terms of addressing the mesh elements, the meshes can be divided into two groups. The first is the group of structured meshes, where the location and connections of a mesh element (cell, edge, vertex, etc.) is determined by its index in the mesh. The second are unstructured meshes. The basic difference between the structured and unstructured mesh is, that the location of an element is determined only by its connections to other elements. In the following sections, we briefly discuss the basic qualities of these types of meshes.

Structured Mesh

A structured mesh is such tessellation of a domain, where the location of a mesh element is determined by its index. By the location we mean the exact position in the space and connections with other cell elements. Figure 1.1 presents few common types of structured meshes. Such meshes are suitable to tessellate simple domains like combination of rectangles in 2D. For example, the domain tessellation presented in Figure 1.1b is suitable for finite element method [32]. It is possible to handle simple mesh refinements using structured mesh as shown in Figure 1.1c. This way it is possible for the mesh to be finer, for example, near the boundary. If the problem requires a more complex domain, structured mesh can be mapped from simpler domain as shown in Figure 1.2.

These types of meshes are not efficient to describe more complex areas, for example, as shown in Figure 1.3. In this case, it would be very complicated to define a mapping from simpler domain. Moreover, the structured mesh is not suitable to store cells of more types, i.e., all cells are triangles or all are rectangles. Therefore, we decided to interest in unstructured mesh.

Unstructured Mesh

An unstructured mesh can be defined as a structure of elements, where explicit knowledge of elements connection is necessary, to determine the location of a mesh element. Before the general unstructured mesh will be introduces, let us discuss a case with predefined cell shapes as cell type.

In order to simplify the data structure and to lower the memory requirements, it is possible to define a set of geometrical primitives. For example, the primitives can be triangle or rectangle in 2D, or tetrahedron, pyramid or hexahedron in 3D. Using this concept it is possible to define a cell with a set of its vertices. The connection of edges or faces to vertices can be derived from the cell vertices and the cell type.

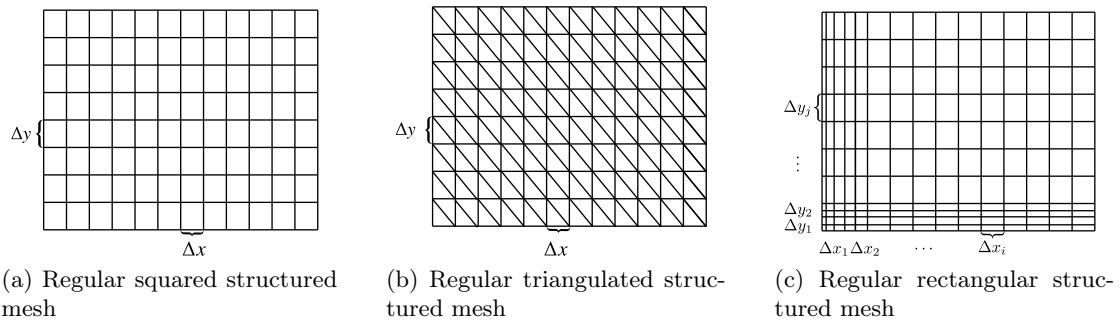


Figure 1.1: Examples of structured meshes

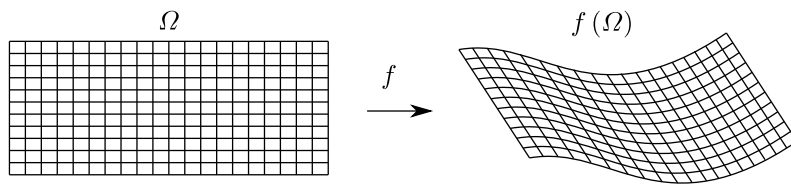


Figure 1.2: A simple computational domain and its mesh can be mapped to another domain.

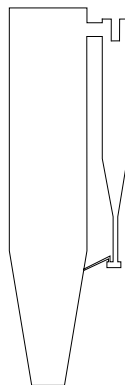


Figure 1.3: Example of a complex computational domain

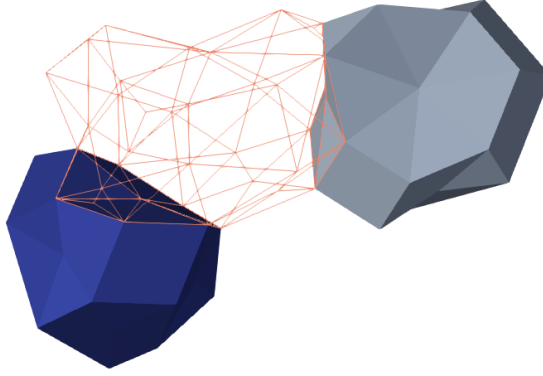


Figure 1.4: Example of general polyhedral cells. The cells had to be split into tetrahedrons because of VTK format capabilities.

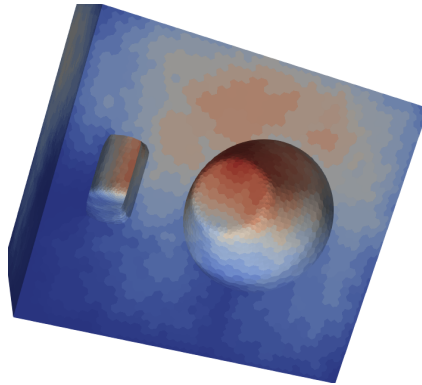


Figure 1.5: Examples of unstructured meshes.

In 2D it is possible to describe any polygon using its vertices, thanks to the topological qualities of edges (faces) in 2D. Thus, this concept is able to represent any mesh in 2D. But in 3D there is no way how to describe a general polyhedron. Thus, it is not possible to represent as complex cell as shown in Figure 1.4. According to [37] such types of non-admissible meshes are more efficient CFD. Therefore, it is worth considering such general types of meshes.

1.1 General Unstructured Polyhedral Meshes

In the introduction of this chapter we mentioned that it is worthy to investigate properties of such generic meshes as shown in Figure 1.4. An advantage of this type of mesh is its ability to tessellate a domain with less cells but richer structure than a structured one. When one completes algorithms for these generic meshes, it is easy to work with locally refined mesh. Moreover, it is possible to refine the mesh according to the needs. All of those advantages sprout from the low limitations on the mesh elements topology.

In the following sections the problem of storing general meshes is thoroughly discussed. Firstly, let us define mesh notation.

1.2 Notation on Unstructured Mesh

In order to develop the numerical scheme of a problem, it is necessary to introduce a division of the spatial domain into a set of disjunctive open subsets of $\Omega \subset R^d$, $d \in \mathbb{N}$. This system is called a mesh or grid. In this work, computation on an unstructured mesh is considered. The mesh is then employed in the numerical method to produce the spatial discretization of given formula,

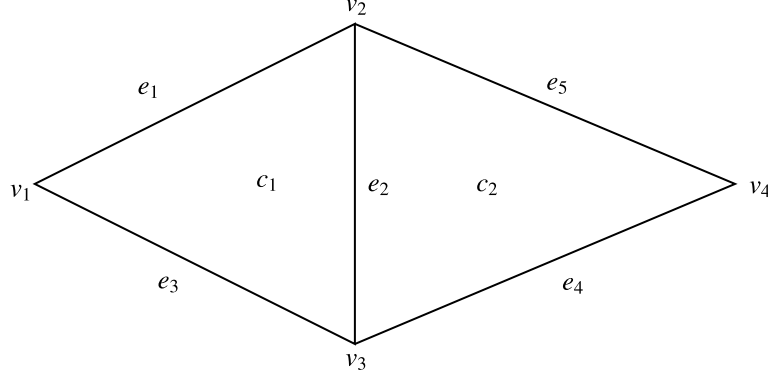


Figure 1.6: Example of connections in a simple 2D mesh. Vertices connected to element c_1 are $\{v_3, v_2, v_4\}$. The cells connected to element v_1 are $\{c_1, c_2\}$.

further described in next section. First, the notation is to be introduced.

Let $A \subset \Omega \subset \mathbb{R}^d$, then $m(A)$ denotes a d -dimensional Lebesgue measure of the set A . Similarly, for $S \subset \Omega \wedge S \subset \mathbb{R}^{d-1}$ $\tilde{m}(S)$ denotes $(d-1)$ -dimensional Lebesgue measure. In order to simplify the notation, from now on the tilde above \tilde{m} will be omitted and the applied function will be clear out of the context.

The mesh is a set of subsets of Ω denoted \mathcal{T} . Elements of \mathcal{T} are called cells aka control volumes. Cells are polygonal or polyhedral open convex subsets of Ω . The \mathcal{E} is then system of $(d-1)$ -dimensional elements constructing cells boundary aka edges or faces.

Definition 1. The mesh \mathcal{T} and \mathcal{E} tessellating the domain $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$ are defined as follows:

1. $\overline{\bigcup_{K \in \mathcal{T}} K} = \bar{\Omega}$.
2. $(\forall K \in \mathcal{T}) (\exists \mathcal{E}_K \subset \mathcal{E}) \left(\partial K = \bigcup_{\sigma \in \mathcal{E}_K} \bar{\sigma} \right)$.
3. $(\forall K, L \in \mathcal{T}) (K \neq L \implies (\tilde{m}(\bar{K} \cap \bar{L}) = 0 \vee (\exists \sigma \in \mathcal{E}) (\sigma = \bar{K} \cap \bar{L})))$.

The notation introduced in Definition 1 is sufficient to describe an unstructured mesh from the mathematical point of view. However, in order to describe the polyhedral mesh geometrically, we have to define basic mesh construction elements. The geometrical construction elements are, for example, cells, faces and vertices, which define the tessellation. In the literature, these elements are also called *entities*.

Definition 2. Let \mathcal{T} be a d -dimensional mesh, where $d \in \mathbb{N}$. The set of elements of dimension k are \mathcal{T}^k , where $k \in \{0, 1, \dots, d\}$. The $N_{\mathcal{T}}^k = |\mathcal{T}^k|$ is the number of elements in of dimension k . Finally, the complete system of geometrical elements is defined as

$$\mathcal{T}^* = \bigcup_{k=0}^d \mathcal{T}^k. \quad (1.1)$$

For certain dimensions, the elements are named as follows:

- The d -dimensional element is a cell. Moreover, according to Definition 1, $\mathcal{T}^d = \mathcal{T}$ holds. In further text, elements of \mathcal{T}^d will be denoted as c_1, c_2, \dots
- The $(d-1)$ -dimensional element is a face. Moreover, according to definition 1, $\mathcal{T}^{d-1} = \mathcal{E}$ holds. In further text, elements of \mathcal{T}^{d-1} will be denoted as f_1, f_2, \dots
- The 1-dimensional element is an edge (line in space), the elements of \mathcal{T}^1 will be denoted as e_1, e_2, \dots

- The 0-dimensional element is a vertex, it is equivalent to $\mathbf{x} \in \Omega$. The elements of \mathcal{T}^1 will be denoted as v_1, v_2, \dots

Note that in 2D mesh, the faces and edges are the same. All elements except vertices are bounded by elements with lower dimension. The vertices have a special position in construction of mesh, because they are the only elements which are directly given position in space. Other elements location is defined by their connections to vertices.

Definition 3. $e, f \in \mathcal{T}^*$ are connected $\iff e \subset \partial f$ or $f \subset \partial e$.

Furthermore, if the elements $e, f \in \mathcal{T}^*$ are connected and the dimension of f is greater than the dimension of element e , then the e is a sub-element of f and conversely f is super-element of e . It is also obvious that two elements of the same dimension can not be connected, because one can not be part of the other's boundary. In terms of connections, it is possible to define a trivial neighborhood.

Definition 4. Let $e \in \mathcal{T}^*$, then the set of all connected elements is denoted

$$N(e) = \{e' \in \mathcal{T}^* \mid e' \text{ is connected to } e\}.$$

Moreover, we denote the subset of the connected elements with the given dimension d as

$$N^d(e) = N(e) \cap \mathcal{T}^d.$$

The concept of connections allows to describe the topological or geometrical neighborhood of elements in the mesh. For example, cells connected to a particular face are neighbors in the sense that they share at least one face. This property will be advantageously used further in this thesis.

Note that the notation using elements vertex, edge, face, cell is sufficient to describe all mesh element occurring in meshes of dimension less or equal to 3. In a generic case, element types are referred by their dimension. This system of elements is able to describe any polyhedral mesh or similar system in any acceptable dimension.

1.3 Data Structure Representing Meshes with Arbitrary Topology

This section describes data structures representing general unstructured meshed and their advantages and disadvantages. It was already mentioned that representation of an unstructured mesh requires explicitly stored connections between elements. However, it is not necessary to store all the connections in the mesh to have full information about the mesh topology. Moreover, the connections do not have to be symmetrical, e.g., it is sufficient when the cell refers to their faces and the faces do not need to point to the cells. The backward connections can be obtained by applying a simple algorithm, which will be discussed below.

1.3.1 Graph Description

To describe the system of connections, we use a directed graph $G_{\mathcal{T}^*} = (V_{\mathcal{T}^*}, E_{\mathcal{T}^*})$, where the vertices of the graph match the elements of the mesh and edges of the graph reflect the connections between elements in the mesh.

The vertices $V_{\mathcal{T}^*}$ of $G_{\mathcal{T}^*}$ are grouped into layers by dimensions of elements:

$$V_{\mathcal{T}^*} \cong \mathcal{T}^* = \mathcal{T}^0 \cup \mathcal{T}^1 \cup \dots \cup \mathcal{T}^d, \quad (1.2)$$

$$V_{\mathcal{T}^*}^k \cong \mathcal{T}^k. \quad (1.3)$$

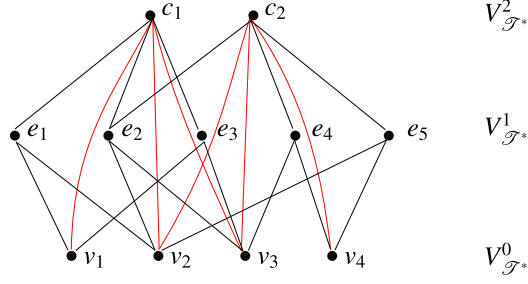


Figure 1.7: Example of graph $G_{\mathcal{T}^*}$ to the mesh shown in Figure 1.6. The edges in the graph are presented by black or red lines. The edges between cells and vertices are highlighted red, for better readability.

The edges of $G_{\mathcal{T}^*}$ are defined as follows,

$$E_{\mathcal{T}^*} = \{ (e, e') \in V_{\mathcal{T}^*}^2 \mid e, e' \text{ are connected} \}. \quad (1.4)$$

For an example see Figure 1.7.

The graph $G_{\mathcal{T}^*}$ contains all connections in the mesh. Since the connections of elements is symmetrical relation, the graph $G_{\mathcal{T}^*}$ is de facto not directed. That means $(e, e') \in E_{\mathcal{T}^*} \iff (e', e) \in E_{\mathcal{T}^*}, \forall e, e' \in V_{\mathcal{T}^*}$. Additionally, for simpler description of connections, i.e., graph edges, we denote a subset of graph edges from dimension d_1 to dimension d_2 as follows

$$E_{\mathcal{T}^*}^{d_1, d_2} = \{ (e, e') \in E_{\mathcal{T}^*} \mid e \in V_{\mathcal{T}^*}^{d_1}, e' \in V_{\mathcal{T}^*}^{d_2} \}. \quad (1.5)$$

In the following analysis of the underlying mesh data structure, adjacency matrix will be used [31].

Definition 5. Let $G = (V, E)$ be a graph. The adjacency matrix of the graph G is matrix $\mathbb{A}_G \in \mathbb{R}^{|V| \times |V|}$ defined as follows:

$$[\mathbb{A}_G]_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{if } (v_i, v_j) \notin E, \end{cases} \quad (1.6)$$

where $i, j \leq |V|, v_i, v_j \in V$.

Furthermore, to investigate certain types of connections and properties of the graph mesh representation, we introduce a connection matrix. The idea of connection matrix is based on adjacency matrix. The connection matrix from dimension d_1 to dimension d_2 reads

$$\left[\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_2} \right]_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, v_i \in V_{\mathcal{T}^*}^{d_1}, v_j \in V_{\mathcal{T}^*}^{d_2}, \\ 0 & \text{if } (v_i, v_j) \notin E, v_i \in V_{\mathcal{T}^*}^{d_1}, v_j \in V_{\mathcal{T}^*}^{d_2}, \end{cases} \quad (1.7)$$

where $\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_2} \in \mathbb{R}^{N_{\mathcal{T}^*}^{d_1} \times N_{\mathcal{T}^*}^{d_2}}$. The connection matrix is de facto a rectangular block of $A_{G_{\mathcal{T}^*}}$. The similarity of notation of the connection matrix and adjacency matrix is visible in Figure 1.8. For example $\mathbb{A}_{G_{\mathcal{T}^*}}^{1,0}$ reflects the connections from edges to vertices, i.e. $E_{\mathcal{T}^*}^{1,0}$. It can be easily seen that the equation

$$\left(\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_2} \right)^T = \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{d_2, d_1} \right), \quad (1.8)$$

holds for any $d_1, d_2 \in \{0, 1, \dots, d\}$. An example of adjacency matrix to the mesh presented in Figure 1.6 is shown in Figure 1.8.

Further in this work, to design the data structure representing the mesh, we intend to study only certain types of connections, i.e., a certain subgraph of $G_{\mathcal{T}^*}$.

	v_1	v_2	v_3	v_4	e_1	e_2	e_3	e_4	e_5	c_1	c_2
v_1	0	0	0	0	1	0	1	0	1	1	0
v_2	0	0	0	0	1	1	0	0	0	1	1
v_3	0	0	0	0	0	1	1	1	0	1	1
v_4	0	0	0	0	0	0	0	1	1	0	1
e_1	1	1	0	0	0	0	0	0	0	1	0
e_2	0	1	1	0	0	0	0	0	0	1	0
e_3	1	0	1	0	0	0	0	0	0	1	1
e_4	0	0	1	1	0	0	0	0	0	0	1
e_5	1	0	0	1	0	0	0	0	0	0	1
c_1	1	1	1	0	1	1	1	0	0	0	0
c_2	0	1	1	1	0	0	1	1	1	0	0

	\mathcal{T}^0	\mathcal{T}^1	\mathcal{T}^2
\mathcal{T}^0	$\mathbb{A}_{G_{\mathcal{T}^*}}^{0,0}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{0,1}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{0,2}$
\mathcal{T}^1	$\mathbb{A}_{G_{\mathcal{T}^*}}^{1,0}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{1,1}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{1,2}$
\mathcal{T}^2	$\mathbb{A}_{G_{\mathcal{T}^*}}^{2,0}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{2,1}$	$\mathbb{A}_{G_{\mathcal{T}^*}}^{2,2}$

$$\mathbb{A}_{G_{\mathcal{T}^*}} =$$

Figure 1.8: Adjacency matrix of the graph representing the example mesh in Figure 1.6, for the graph representation see Figure 1.7. It is possible to write the adjacency matrix as a block matrix using the connection matrices, see the right representation. In the left matrix the blocks are separated with dashed lines.

1.3.2 Possible Representations

In this section, some of the data structures representing \mathcal{T}^* are discussed. As introduced in the previous Section 1.3.1, the topology of a mesh \mathcal{T}^* can be described using the graph $G_{\mathcal{T}^*}$.

In this work, the main purpose of the mesh is solving problems for PDE's using finite volume method [36, 42, 45, 30]. Basic properties of this method are:

- repeated accesses to mesh elements during computation,
- possible use of generic geometrical elements e. g. polygons and polyhedrons.

To satisfy the needs, the data structure must:

- contain all topology information about the mesh \mathcal{T}^* . (It is equivalent to the possibility of reconstruction of $G_{\mathcal{T}^*}$ or $\mathbb{A}_{G_{\mathcal{T}^*}}$),
- allow instant access to grid elements and its connected elements,
- efficiently store grid information with respect to cache usage,
- contain the information about the face or edge orientation and its normal vector,
- be able to store a polyhedral mesh with generic topology,
- be able to add or remove grid elements.

In order to make the data structure efficient for working with the mesh, it is necessary to avoid dynamic memory allocation as much as possible and prevent runtime operations. Thus, more sophisticated data structures as a hash map are not suitable to solve this problem. Detailed description of the implementation is in Chapter 2.

For the description of the data structures, a 3D polyhedral mesh will be considered. As discussed in Section 1.3.1, storing all connections of the graph $G_{\mathcal{T}^*}$ is very expensive, inefficient and moreover, not necessary. Thus, we aim to reduce the amount of stored information by omitting some edges in $G_{\mathcal{T}^*}$. So the problem is to find a subgraph

$$\tilde{G}_{\mathcal{T}^*} = (V_{\mathcal{T}^*}, \tilde{E}_{\mathcal{T}^*}), \quad (1.9)$$

where $\tilde{E}_{\mathcal{T}^*} \subset E_{\mathcal{T}^*}$, with qualities meeting the above requirements. The main goal is to be able to reconstruct $G_{\mathcal{T}^*}$ from $\tilde{G}_{\mathcal{T}^*}$. This can be achieved, for example, using relation (1.8) and

Cell {	Face {	Edge {	Vertex {
idFace [n]	idEdge [n]	idVert [2]	coordinates [3]
}	}	}	}

Figure 1.9: Data structure scheme of the simplest representation in 3D mesh. The structures contains references to other elements, e.g the cell refers to its faces (`idFace[n]`). The number in the square brackets symbolizes the number of references. If the number is not generally known, n is written instead. The vertices does not have any sub-elements, however they contain the coordinates of their position in the space (`coordinates[3]`, i.e. x, y, z).

combinations of connection information:

$$\begin{aligned} \left[\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_2} \right]_{ij} &= \text{connect} \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_3}, \mathbb{A}_{G_{\mathcal{T}^*}}^{d_3, d_2} \right) \\ &= \begin{cases} 1 & \text{if } \left(\exists k \in \{1, 2, \dots, N_{\mathcal{T}^*}^{d_3}\} \right) \left(\left[\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_3} \right]_{ik} \left[\mathbb{A}_{G_{\mathcal{T}^*}}^{d_3, d_2} \right]_{kj} = 1 \right), \\ 0 & \text{else,} \end{cases} \end{aligned} \quad (1.10)$$

where the dimensions d_1, d_2, d_3 satisfy $(d_1 > d_3 > d_2) \vee (d_1 < d_3 < d_2)$. The condition for the dimensions of the connection matrices consists in finding the correct paths in the graph that are consistent with the mesh topology. The problem can be demonstrated on the example mesh shown in Figures 1.6 and 1.8. The connection matrix

$$\mathbb{A}_{G_{\mathcal{T}^*}}^{2,1} = \text{connect} \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{2,0}, \mathbb{A}_{G_{\mathcal{T}^*}}^{0,1} \right)$$

contains connections of cell c_1 to edges e_1, e_2, e_3, e_4, e_5 . This result is incorrect, because the cell c_1 is connected to edges e_1, e_2, e_3 . The problem lies in the joined information, because we got edges neighboring to cells over vertices instead of connected edges. Basically, the connecting algorithm is an algorithm for finding paths of length 2 in the graph. This property of this algorithm will be used in the neighborhood algorithm in Section 2.4.3.

Obviously it is necessary to store either connections from $V_{\mathcal{T}^*}^k$ to $V_{\mathcal{T}^*}^{k-1}$ or from $V_{\mathcal{T}^*}^{k-1}$ to $V_{\mathcal{T}^*}^k$, because those connections cannot be correctly obtained from (1.10).

Basic Representation

The simplest and most native representation of a 3D mesh \mathcal{T}^* stores the following connections

$$\tilde{E}_{\mathcal{T}^*} = E_{\mathcal{T}^*}^{3,2} \cup E_{\mathcal{T}^*}^{2,1} \cup E_{\mathcal{T}^*}^{1,0}. \quad (1.11)$$

The choice of $\tilde{E}_{\mathcal{T}^*}$ means that following connection matrices $\mathbb{A}_{G_{\mathcal{T}^*}}^{3,2}, \mathbb{A}_{G_{\mathcal{T}^*}}^{2,1}, \mathbb{A}_{G_{\mathcal{T}^*}}^{1,0}$ are available. The data structure realizing this representation is presented in Figure 1.9. This representation contains the full information about the mesh topology. For example, that it is possible to reconstruct the topology information, we present how to obtain several types of connections that are not stored:

$$\mathbb{A}_{G_{\mathcal{T}^*}}^{3,1} = \text{connect} \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{3,2}, \mathbb{A}_{G_{\mathcal{T}^*}}^{2,1} \right), \quad (1.12)$$

$$\begin{aligned} \mathbb{A}_{G_{\mathcal{T}^*}}^{0,3} &= \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{3,0} \right)^T \\ &= \left(\text{connect} \left(\mathbb{A}_{G_{\mathcal{T}^*}}^{3,1}, \mathbb{A}_{G_{\mathcal{T}^*}}^{1,0} \right) \right)^T. \end{aligned} \quad (1.13)$$

Although, this representation is able to store the mesh, it is not suitable for application on computer, because the vertices in V^3 have unbounded degree. The next representation solves this problem wisely using topological quality of faces.

Cell { }	Face { idCell [2] idEdge [n] }	Edge { idVert [2] }	Vertex { coordinates [3] }
----------	---	---------------------------	----------------------------------

Figure 1.10: Data structure scheme of the representation with reverted connections between cells and faces. The notation is the same as in Figure 1.9. Note that the number of face references to cells is 2 independently on the mesh, i.e., `idCell [2]`.

Cell { idFace [n] }	Face { idEdge [n] }	Edge { idVert [2] }	Vertex { idCell [n] coordinates [3] }
---------------------------	---------------------------	---------------------------	--

Figure 1.11: Data structure scheme of the representation chosen in [28].

Representation with Reverted Cell to Face Reference

Thanks to the fact that one face can be connected to up to two cells, it is possible to improve the previous scheme. The chosen connections are:

$$\tilde{E}_{\mathcal{T}^*} = E_{\mathcal{T}^*}^{2,3} \cup E_{\mathcal{T}^*}^{2,1} \cup E_{\mathcal{T}^*}^{1,0}. \quad (1.14)$$

This representation is very similar to the previous one. Thus, this representation also satisfies the requirement on the reconstruction of $G_{\mathcal{T}^*}$. The data structure realizing this representation is shown in Figure 1.10. The subtle difference consisting in reverting the connections between cells and faces has a very significant effect. This way, the degree of the vertices in V^2 are bounded by 2 in the subgraph $\tilde{G}_{\mathcal{T}^*}^{3,2}$. Therefore, this representation is more suitable for implementation, because it requires less dynamically allocated memory. The face data structure can be optimized and the number of cells can be preset to 2. This optimizes cache usage. Unfortunately, a similar trick does not work for connections $E_{\mathcal{T}^*}^{2,1}$ and the degree of faces in the graph $\tilde{G}_{\mathcal{T}^*}$ remains unbounded.

Moreover, storing the connections from faces to cells has a particular advantage for FVM as will be noted further in Section 5.2.2.

Comparison with Other Published Works

Beall [28] presents similar description of mesh to ours. However, they require at least one cycle in the graph $\tilde{G}_{\mathcal{T}^*}$, in order to simplify some operations and iteration on the mesh. Therefore, the selected representation is

$$\tilde{E}_{\mathcal{T}^*} = E_{\mathcal{T}^*}^{3,2} \cup E_{\mathcal{T}^*}^{2,1} \cup E_{\mathcal{T}^*}^{1,0} \cup E_{\mathcal{T}^*}^{0,3}. \quad (1.15)$$

This representation is the same as (1.11), because it is possible to obtain $\mathbb{A}_{G_{\mathcal{T}^*}}^{0,3}$ according to (1.13). Thus, this representation is worse than the basic one presented as the first, because it enforces storing $E_{\mathcal{T}^*}^{0,3}$ as $\left(\mathbb{A}_{G_{\mathcal{T}^*}}^{0,3}\right)$ directly in the mesh data structure which breaks space locality, while it is possible to compute the $\mathbb{A}_{G_{\mathcal{T}^*}}^{0,3}$ on the fly. If we do not need the connections from vertices to cells, then the $\mathbb{A}_{G_{\mathcal{T}^*}}^{0,3}$ is not needed at all and there is no reason to store them.

1.3.3 Representation of Choice

The chosen representation is based on the representation in (1.14). Because the connections between cells and faces are reverted, the iteration over boundary of a cell is more complicated and requires the creation of $\mathbb{A}_{G_{\mathcal{T}^*}}^{3,2}$. To solve this problem, the combination of the fact that a face has up to 2 neighboring cells and the concept of linked list data containers could be used. The

```

Cell {
  idBoundaryFace
}
Face {
  idCellLeft
  idCellRight
  idNextFaceToCellLeft
  idNextFaceToCellRight
  idEdge[n]
}
Edge {
  idVert[2]
}
Vertex {
  coordinates[3]
}

```

Figure 1.12: The scheme of the chosen representation on unstructured mesh.

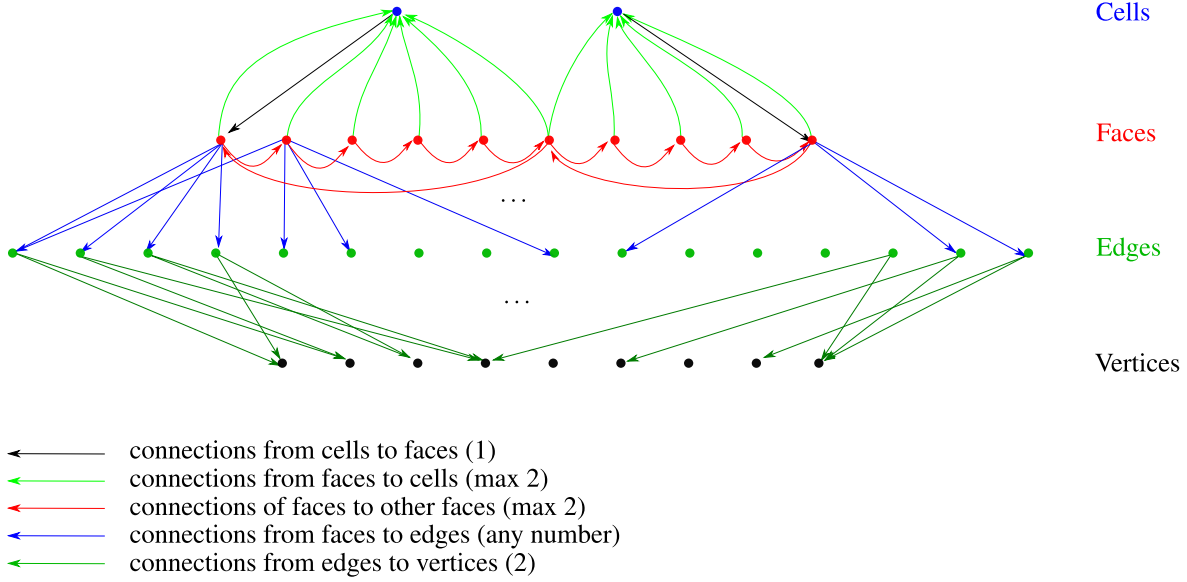


Figure 1.13: Graph scheme of the chosen representation in 3D case.

point is to chain the boundary of a cell as a cycled uni-directional linked list. This is reached by adding a reference to the next boundary element for both right and left cells in the face data structure. Now, in order to allow direct iterations over cell boundary, it is sufficient to add a single reference in the cell pointing to any of the cell boundary faces. The scheme of current data structure is shown in Figure 1.12.

The graph representing the final data structure reads $G_{\mathcal{T}^*}^* = (\mathcal{T}^*, E_{\mathcal{T}^*}^*)$, where

$$E_{\mathcal{T}^*}^* = E_{\mathcal{T}^*}^{*3,2} \cup E_{\mathcal{T}^*}^{*2,2} \cup E_{\mathcal{T}^*}^{2,3} \cup E_{\mathcal{T}^*}^{2,1} \cup E_{\mathcal{T}^*}^{1,0}, \quad (1.16)$$

where $E_{\mathcal{T}^*}^{*3,2}, E_{\mathcal{T}^*}^{*2,2}$ are newly added connections. For easier understanding, see Figure 1.13, where the connections $E_{\mathcal{T}^*}^{*3,2}$ are marked by black arrows, and $E_{\mathcal{T}^*}^{*2,2}$ are highlighted in red. Note that in terms of graph theory, the graph must contain cycles in $E_{\mathcal{T}^*}^{*2,2}$ with respect to connections $E_{\mathcal{T}^*}^{2,3}$.

Compared to the representation (1.14), this representation allows direct iteration of element boundary while still having bounded degrees of $V_{\mathcal{T}^*}^3, V_{\mathcal{T}^*}^1, V_{\mathcal{T}^*}^0$ and the $V_{\mathcal{T}^*}^2$ has bounded number of references to cells. Therefore, this representation is very promising to work efficiently on common processor architectures.

2D Example

In the case of two dimensional mesh, our representation benefits from the fact that face element coincides with edge. Therefore, the stored connections in graph $G_{\mathcal{T}^*}^* = (\mathcal{T}^*, E_{\mathcal{T}^*}^*)$ are the following:

$$E_{\mathcal{T}^*}^* = E_{\mathcal{T}^*}^{*2,1} \cup E_{\mathcal{T}^*}^{*1,1} \cup E_{\mathcal{T}^*}^{1,2} \cup E_{\mathcal{T}^*}^{1,0}. \quad (1.17)$$

The system of connections might be better seen from Figure 1.15.

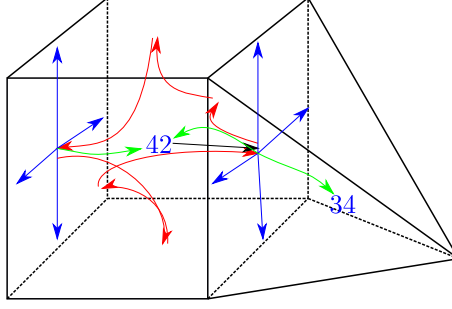


Figure 1.14: Example of connection on 3D mesh corresponding to the graph presented in Figure 1.13.

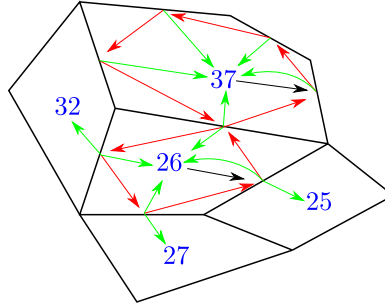


Figure 1.15: An example of connections in a 2D mesh

The biggest advantage of our representation in 2D is that the data structure is static while it is still able to store any polygonal mesh.

d-dimensional Example

In the most generic case, i.e., the dimension of the mesh is greater than 3. The chosen representation $G_{\mathcal{T}^*}^* = (\mathcal{T}^*, E_{\mathcal{T}^*}^*)$ is the following:

$$E_{\mathcal{T}^*}^* = E_{\mathcal{T}^*}^{*d,d-1} \cup E_{\mathcal{T}^*}^{*d-1,d-1} \cup E_{\mathcal{T}^*}^{d-1,d} \cup E_{\mathcal{T}^*}^{d-1,d-2} \cup E_{\mathcal{T}^*}^{d-2,d-3} \cup \dots \cup E_{\mathcal{T}^*}^{2,1} \cup E_{\mathcal{T}^*}^{1,0}. \quad (1.18)$$

For easier understanding of the generic concept, see Figure 1.16. Thanks to the properties of named elements in Definition 2, it is possible to perform the same trick as in the 3D case. The faces has up to 2 neighboring cells independently on the mesh dimension. The connections $E_{\mathcal{T}^*}^{*d,d-1}$, $E_{\mathcal{T}^*}^{*d-1,d-1}$, $E_{\mathcal{T}^*}^{d-1,d}$ are again marked by black, red and green arrows in Figure 1.16 as well as in previous 3D case, Figure 1.13.

In this case as a generalization of 3D case, there are more layers of elements in the graph that are connected to others by blue arrows. Those are the connections $E_{\mathcal{T}^*}^{d-1,d-2}$, $E_{\mathcal{T}^*}^{d-2,d-3}$, \dots , $E_{\mathcal{T}^*}^{2,1}$. As was already mentioned, storing those connections is expensive, but it must be seen as a tax for having unstructured mesh with arbitrary topology in generic dimension. Moreover, as a consequence of the connection of adjacency matrices (1.10) (combination of connections), there is no way how to develop a data structure without storing those connections represented by blue arrows.

Reserved Number of Sub-Elements

Unfortunately, there is no way, to bound degrees in connections from faces to edges while still allowing arbitrary mesh topology. There is possible only limited solution, to prescribe the maximum number of references represented by blue arrows in Figure 1.16. Moreover, the limit can be

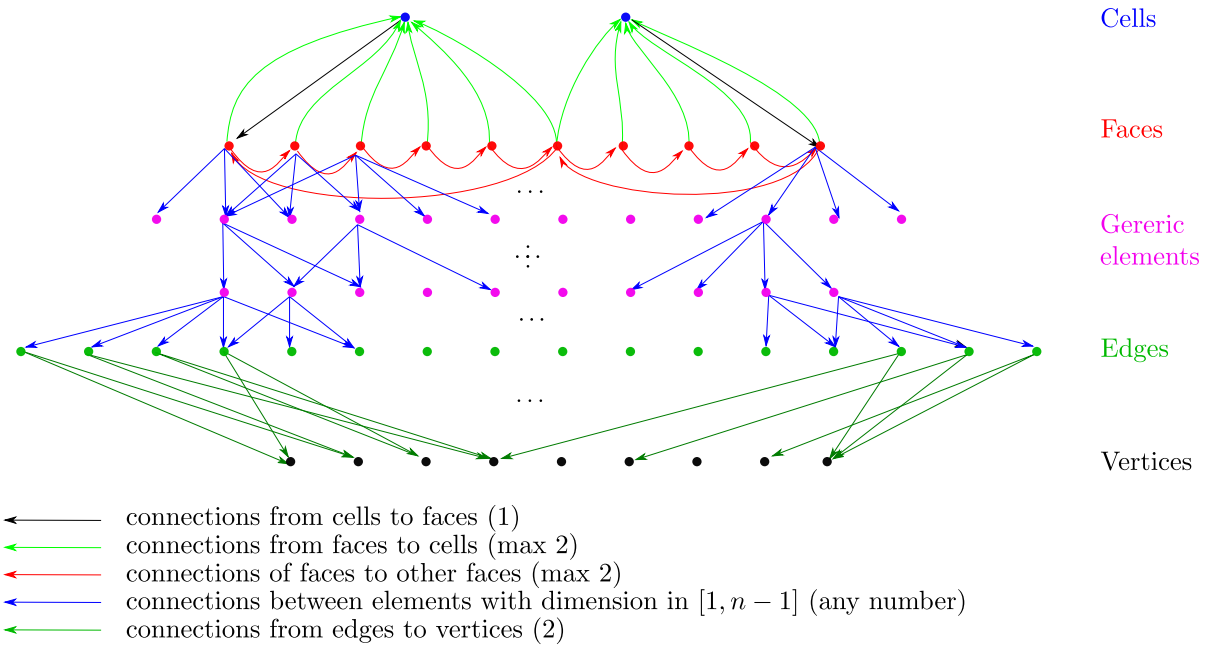


Figure 1.16: This is an example of a graph representing a generic topology mesh in generic dimension. In comparison to 3D case in Figure 1.13, a violet layer representing generic elements appears here. These generic elements have dimension $d \in [2, n - 1]$ and they refer to their boundary using blue arrows $E_{\mathcal{T}^*}^{d, d-1}$. Fortunately, other elements preserve their qualities.

prescribed for every dimension separately. For example, if it is set as $n \leq 3$ in a 3D mesh, then all the faces have to be triangles. The higher the bound for n , the richer mesh topology is allowed. The advantage of prescribing the maximum number is, that the data structure can be optimized and the indexes from faces to edges can be embedded into the face structure. Therefore, dynamic allocation of memory in the structure is prevented altogether.

Chapter 2

The GTMesh Library

This chapter describes the construction and implementation of a C++ template library for working with unstructured mesh with a general topology in an arbitrary dimension. The implementation is based on the data structure presented in Section 1.3.3. The discussion in this chapter does not focus only on the description of the final library, but it aims to introduce the whole process of developing such a complex project with all the architecture, concepts and know how.

The implementation of GTMesh (general topology mesh) utilizes modern C++ techniques as template metaprogramming, SFINAE, and ADL [52, 3, 5]. The template metaprogramming denotes techniques of automatic generation of temporary code based on template parameters. SFINAE (Substitution Failure Is Not An Error) is applied, e.g., when the compiler looks for the candidates to the implementation of the given function or class and the parameter setup causes an error, then the candidate is ignored, but the compilation continues with another candidate. ADL (Argument Dependent Lookup) is a set of rules of looking up of unqualified names, e.g., enables to utilize a later definition of an overloaded function. The used standard was held down at C++14 except one optimization for C++17 (usage of `string_view` for hash calculation). Moreover, the realization of the GTMesh library required creation of other interesting tools useful for debugging and data IO, which are described in Chapter 4.

Firstly, let us describe the vision of the final product and corresponding project architecture. Without the effort put in the development of the project architecture, we might not be able to achieve such results.

2.1 Project Architecture

The aim of GTMesh library is to provide framework for working with a general unstructured mesh in an intuitive and unrestricted way. In the case of developing the GTMesh library, it helped us to start from the vision of the final product. Therefore, for the sake of better understanding the implementation of the internals we start with the description of what exactly is expected from the class `UnstructuredMesh`, which provides the user interface to most of GTMesh features. We expect from the `UnstructuredMesh` class to provide:

1. a simple definition, where everything can be set by template parameters (see code listing 2.1),
2. the possibility to simply access the mesh elements and iterate over them the mesh (see code listing 2.2),
3. basic functionalities such as measure computation or connections reconstruction (see code listing 2.3),
4. the possibility to map data onto mesh elements easily (see code listing 2.4),

Code listing 2.1 Example of a definition of UnstructuredMesh. The definition must specify the mesh dimension, reference types, type of coordinates and finally reserves for numbers of references as described in Section 1.3.3.

```

1 // 2D unstructured mesh size_t references and double precision coordinates
2 UnstructuredMesh<2, size_t, double> mesh2D;
3
4
5 // 3D unstructured mesh size_t references and double precision coordinates
6 // with maximum nuber of face sub-elements prescibed to 6
7 UnstructuredMesh<3, size_t, double, 6> mesh3D;
8
9 // with dynamically allocated sub-elements of faces,
10 // i.e. face could have any number of edges
11 UnstructuredMesh<3, size_t, double> mesh3D;
12
13
14 // 5D unstructured mesh size_t references and double precision coordinates
15 // maximum number of cell boundary sub-elements prescibed to 4
16 // maximum number of sub-elements of elements of dimension 3 prescibed to 4
17 // maximum number of sub-elements of elements of dimension 2 prescibed to 6
18 UnstructuredMesh<5, size_t, double, 4, 4, 6> mesh5D;

```

Code listing 2.2 Example of working with mesh elements and the mesh itself.

```

1 UnstructuredMesh<3, size_t, double, 6> mesh;
2 // obtaining the cell with index 0
3 mesh.getElements<3>().at(0);
4 mesh.getCells().at(0);
5
6 // otaining the vertex on index 0
7 mesh.getElements<0>().at(0);
8 mesh.getVertices().at(0);
9
10 // iteration over boundary of cell 0
11 auto& cell_0 = mesh.getCells().at(0);
12 size_t bElemIndex = cell_0.getBoundaryElementIndex();
13 do {
14     // ... do some stuff
15     // get next boundary element index
16     bElemIndex = mesh.getFaces().at(bElemIndex).getNextBElem(cell_0.getIndex());
17 } while (bElemIndex != cell_0.getBoundaryElementIndex());
18
19 // or equivalently
20 mesh.apply<3, 2>(
21     0,
22     [](size_t cellIndex, size_t faceIndex){
23         // ... do some stuff
24     });
25
26
27
28 // moreover it is possible to perform deeper loops
29 // and for all elements
30 mesh.apply<3, 0>(
31     [](size_t cellIndex, size_t faceIndex){
32         // ... do some stuff
33     });

```

Code listing 2.3 Example of generic functions that are usually applied to the mesh. The point is to automatically apply the correct implementation of a function according to mesh template arguments. The aim is to let the user have feeling that there is no richer structure behind the `UnstructuredMesh` structure. Thanks to this, the library is user friendly and easy to work with.

```

1 // connections from vertices to cells
2 auto vertexToCellConnection = mesh.connections<0,3>();
3
4 // obtain respective Lebesgue measure of all elements of the mesh
5 // with dimension greater than 0
6 auto measures = mesh.computeElementMeasures();
7
8 // obtain vectors perpendicular to faces
9 auto normals = mesh.computeFaceNormals();
10
11 // calculate the centers of all elements with dimension greater than 0
12 auto centers = computeCenters<DEFAULT>(mesh);

```

Code listing 2.4 Mapping data to mesh elements. The data shall be allocated on each dimension separately and it should look like the data are allocated in the mesh, while the container is completely selfstanding. This concept is very important for implementation of functions as the automatic and generic result type.

```

1 // obtain respective Lebesgue measure of all elements of the mesh
2 // with dimension greater than 0
3 auto measures = mesh.computeElementMeasures();
4
5 // the length of the edge with index 0
6 measures.getDataByDim<1>()[0];
7
8 // the surface area of the face with index 0
9 measures.getDataByDim<2>()[0];
10
11 auto& cell_0 = mesh.getCells().at(0);
12 // the volume of the cell 0
13 measures[cell_0];
14
15 dists = mesh.computeCellsDist();
16
17 MeshDataContainer<std::tuple<double, float>, 3, 2> data;
18 data.allocateData(mesh);
19
20 for(auto cell& : mesh.getCells()) {
21     data.at(cell) = 1.0 / measures.at(cell);
22 }
23
24 data.allocateData(mesh);
25 for(auto face& : mesh.getFaces()) {
26     data.at(face) = measures.at(face) / dists.at(face);
27 }

```

Code listing 2.5 A concept of a convenient interface to data import and export.

```

1 UnstructuredMesh<3, size_t, double, 6> mesh;
2 // load the mesh from fpma file
3 FPMAMeshReader<3> reader;
4 ifstream file("MeshFile.fpma");
5 reader.loadFromStream(file, mesh);
6
7 // export the mesh into VTK file
8 VTKMeshWriter<3, size_t, double> writer;
9 ofstream ofile("MeshFile.vtk");
10 writer.writeHeader(ofile, "fpma_output");
11 writer.writeToStream(ofile, mesh);

```

5. methods for data and mesh import and export (see code listing 2.5).

As the requirements on GTMesh are specified, we can proceed with the project architecture.

The aim is to create a code as generic as possible that would automatically adapt the data structures to the particular setup. This can be achieved by utilizing modern C++ template programming techniques such as templates specializations, conditional inheritance, template metaprogramming etc. Moreover, this approach benefits from more compact and efficient code, which resolves in less lines of code and as a consequence, less errors.

The first challenge is to design a structure to store the unstructured mesh. The mesh consists of elements which properties depend on their dimension. The structure of the mesh elements is based on the structure in Figure 1.12, but we intend to add some attributes for programming purposes. Moreover, certain properties are dependent on mesh dimension, e.g., how a cell refers to one of its faces or a face connects cells. Hence, the plan is to prepare those properties as separate structures. Then, they will be inherited by the mesh element depending on its template arguments. Moreover, as was already discussed, we plan to optimize the structure to prevent dynamic allocation of memory. Therefore, we want to prescribe a reserve for the array of references to sub-elements, i.e., the blue arrows in the Figure 1.16. Using the reserve, it is possible to embed the references directly in the structure of an element. The final structure representing a mesh element is called `MeshElement`.

Once the `MeshElement` structure is implemented, the next step is to gather the `MeshElement` structures in a structure called `MeshElements` as explained in detail in Section 2.2.4. `MeshElements` is the most fundamental structure of the project, since it contains all the mesh elements with their connections. The `MeshElements` structure will utilize tuple like implementation to automatically create containers for respective `MeshElement` structures according to the template arguments.

Then we create global functions working with the `MeshElements`. Thanks to the fact, that the functions are defined outside of the `MeshElements`, it is easier to write compact and generic code, because the function can be implemented in its own header file. Moreover the functions plan to utilize template specializations, which are limited when defined in another class. Therefore, it is easier to make functions automatically respect the mesh dimension. For example, calculation of elements centers or measures, where the computation formula depends on mesh dimension. The same concept of programming can be used for import and export of the `MeshElements`.

However, there occurs a problem bound to the implementation of the functions working with the mesh. The problem is, what is the result type of such function. For example, the function calculating the measures of elements of the mesh, the result must be mapped to all dimensions greater than zero. This problem is solved by developing a special container, that maps data to the mesh to each dimension separately. This way, it is capable to map data automatically to all dimensions higher than zero, e.g., measures of elements. This container is called `MeshDataContainer` and has a significant impact on user-friendliness of the library.

Finally, as algorithms and functions on the `MeshElements` are done, it is possible to create a simple wrapper called `UnstructuredMesh`. This wrapper only inherits the `MeshElements` structure and thanks to the global implementation of mesh functions, it might offer most of the functionality as its own methods. The realization of such a method is very simple, by applying the corresponding global function on `*this`, because `UnstructuredMesh` is a child of `MeshElements`.

This way it is possible to build the project in a modular way by dividing the implementation of its functionality into separate source files. If we did not consider this approach, all these functions with the implementation in separate template classes would have to be a part of the `MeshElements` structure. In such a case, the header file with the definition of `MeshElements` would have thousands of lines and it would be very difficult to manage.

Let us begin with detailed description of the `MeshElement` and `MeshElements` classes.

```

Connections<Edim = Mdim> {
    idBoundaryFace
}

Connections<1 < Edim < Mdim, Reserve = 0> {
    idSubelement[n]
}

Connections<1 < Edim < Mdim, Reserve > 0> {
    idSubelement[Reserve]
}

MeshElement<Edim = Mdim> {
    elementIndex
    flag
    centerVertex
    Connections<Mdim>
}

MeshElement<Edim = Mdim - 1, Reserve> {
    elementIndex
    flag
    centerVertex
    idCellLeft
    idCellRight
    idNextFaceToCellLeft
    idNextFaceToCellRight
    Connections<Edim, Reserve>
}

Connections<Edim = 1 && Edim < Mdim> {
    idVert[2]
}

Connections<Edim = 0> { }

MeshElement<Edim = 1 && Edim < Mdim - 1> {
    elementIndex
    Connections<1>
}

MeshElement<Edim = 0> {
    elementIndex
    coordinates[3] // i.e. x, y, z
}

```

Figure 2.1: The `MeshElement` structure attributes and specializations. The colors of references matches the references in Figure 1.16. Moreover, we provide two additional properties to the `MeshElement` structure. The first is `elementIndex`, which reflects the position of the element in the array in `MeshElements`. The second is a flag and `centerVertex` highlighted in orange for a cell ($\text{Edim}=\text{Mdim}$) and face ($\text{Edim}=\text{Mdim}-1$). The flag is designed for the user to store simple labels directly in the mesh structure.

2.2 The MeshElements Structure

According to the previous Section 2.1, the `MeshElements` structure is the most important structure of the project because it contains all information about the mesh. The class `MeshElements` consists of $d_{\mathcal{T}} + 1$ arrays of `MeshElement` classes, where $d_{\mathcal{T}} \in \mathbb{N}$ is the mesh dimension. Additionally, according to the vision presented in Code listing 2.1, the structure `MeshElements` has the following template arguments:

1. **MeshDimension**: the geometrical dimension of the mesh, i.e. $d_{\mathcal{T}}$,
2. **IndexType**: integer type of references to elements in the mesh,
3. **Real**: floating point type of coordinates, e.g., `float` or `double`,
4. **Reserve**: variadic template argument of type `unsigned int`, specifies the reserved number of references to sub-elements of elements with dimension between 1 and $d_{\mathcal{T}}$.

These arguments are further passed to the `MeshElement` structure template, the construction of which respects those arguments. To be able to create the `MeshElements` class, we have to start with the construction of basic mesh elements, i.e., the `MeshElement` class template.

2.2.1 Scheme of the MeshElement structure

The name `MeshElement` is a common name for several specializations of the class with different properties and attributes based on the element dimension and mesh dimension. The mesh

elements have the following attributes according to the dimension.

1. The cell element has only one reference to one of its boundary elements.
2. The face element has two references to the left and right cells and two references to the next face of the boundary of both connected cells.
3. The face and cell elements have additional attributes: flag and center vertex.
4. All elements with dimension between 1 and $d_{\mathcal{T}}$ have a prescribed number of references to their sub-elements.
 - (a) If the prescribed number of references is positive, then the references are embedded directly in the structure.
 - (b) If the number of references is equal to 0, then the references are allocated dynamically.
5. The edge element has two references to vertices.
6. The vertex element has its coordinates.
7. All elements have their own index representing their position in the respective array contained in the `MeshElements` structure.

The scheme of the specializations of the structure `MeshElement` is shown in Figure 2.1. As noted in Section 2.1, there are some properties that depend on the element dimension with respect to mesh dimension. We start with the discussion about the implementation of the generic properties of `MeshElement`.

2.2.2 Preparation of Generic MeshElement Properties

Because of the complexity of the `MeshElement` structure, we decided to create those properties as a separate structures. Thanks to the creation and inheriting these attributes as separate structures, we are fulfilling the DRY (don't repeat yourself) programming concept. Specifically, the properties are:

- connection of faces to cells,
- flag and center vertex,
- element index.

Let us start from the basic and most elementary one, `elementIndex` provided by `MeshElementBase` class, which is the base class of the `MeshElement` class.

MeshElementBase The `MeshElementBase` class (see Code listing 2.6) provides an index mandatory for all mesh elements. Every element must have an `elementIndex` referring to itself in the array in which it is contained. Moreover, in the `MeshElementBase` there is an attribute named `globalIndex`. This index is meant to be used in case of decomposition of the mesh, to be able to reconstruct the original mesh from the partial ones.

Cell Connection and Cell Boundary Connection Next, we introduce abstract properties such as `CellConnection`, `CellBoundaryConnection` shown in Code listings 2.7 and 2.8. These structures are designed to be inherited by the mesh element with dimension equal to $d_{\mathcal{T}} - 1$. These structures provide references highlighted in red and lime in Figure 2.1. Moreover, they provide methods simplifying the work with accessing of mesh elements such as neighboring cells or cell boundary. For instance, the methods are:

- getting the reference to the neighbor of the given cell,
- getting reference to the next boundary element with respect to the given connected cell.

Code listing 2.6 MeshElementBase, the base class of MeshElement. The MeshElementBase provides two indexes. The elementIndex is a reference of a MeshElement to itself in the respective array contained in the MeshElements structure. The globalIndex is an index referring to the element in global context, when the mesh is decomposed.

```

1  template<typename IndexType>
2  class MeshElementBase{
3
4      // Index of the element in the mesh or in mesh component
5      IndexType ElementIndex;
6
7      // Global index of element in the mesh component
8      IndexType GobaElementIndex;
9  public: //...
10 };

```

Code listing 2.7 The CellConnection structure provides two indexes to connected cells. This structure is to be inherited by MeshElement class representing the face. The name CellConnection reflects the fact that faces are connected to two cells if they represent their common boundary (see Definition 3). This way cells are connected. This type of connection in the mesh is marked by light green arrows in Figure 1.16.

```

1  template<typename IndexType>
2  class CellConnection {
3      // Indexes to two cells which are connected
4      // with this element
5      IndexType CellRightIndex, CellLeftIndex;
6  public:
7      //... other methods working with the above indexes
8  };

```

Vertex From the construction point of view, the MeshElement with dimension 0 is a vertex. In the further applications, it will be useful to work with the vertex as a vector of coordinates. Thence, we define a class Vertex with vector operations, e.g., element-wise addition or scalar product. From the definition of the Vertex class in Code listing 2.9, the Vertex is de facto `std::array` with additionally defined vector operations. Thanks to the static dimension of the Vertex, it is possible to realize all operations with the Vertex class using template metaprogramming, completely avoiding the use of for loops. An example of inline addition is shown in Code listing 2.10. The motivation of making the operations inline is to optimize the computation, since the compiler can better understand the code and apply hardware-dependent optimization such as superscalar execution. Finally, the vertex mesh element can be implemented by inheriting both Vertex with the given real type and number of coordinates, and MeshElementBase. This way, the MeshElement with dimension 0 is both a Vertex and a MeshElement by the terms of C++.

Computationally Significant Element The last attribute is named computationally significant element and is realized by class named ComputationallySignificantElement, see Code listing 2.12. Computationally significant elements are cells and faces, i.e., the elements directly manipulated by the numerical scheme. This structure provides a label named flag to its child object. Furthermore, it embeds a center vertex which is necessary for the computation of mesh properties such as measures of elements or normal vectors of faces.

2.2.3 Definition of the MeshElement Structure

Thanks to the attributes prepared in Section 2.2.2, it is possible to easily construct the class MeshElement with all specializations dependent on the element and mesh dimensions.

Code listing 2.8 The `CellBoundaryConnection` structure definition. This class inherits the properties of `CellConnections`, because it has to map the indexes to the next boundary elements with respect to connected cells. By having the connections to the cells, it is possible to create such methods as `getNextBElem`, which returns the index to the next element according to the given cell index.

```

1  template<typename IndexType>
2  class CellBoundaryConnection : public CellConnection<IndexType> {
3
4      // Index of the next boundary element with respect to the left cell
5      IndexType NextBElemWRTCL;
6
7      // Index of the next boundary element with respect to right cell
8      IndexType NextBElemWRTCR;
9  public:
10     // ... methods working with above references
11     IndexType getNextBElem(IndexType cellIndex) const;
12 };
13
14
15 template<typename IndexType>
16 IndexType CellBoundaryConnection<IndexType>::getNextBElem(IndexType cellIndex) const{
17
18     // If cell is invalid then throw an exception
19     if (cellIndex == INVALID_INDEX(IndexType)) {
20         throw std::runtime_error("Invalid index given to the getNextBElem");
21     }
22
23     // If the cell is equal to Cell1 then return the NextBElemWRTCR
24     if (cellIndex == this->getCellRightIndex()){
25         return getNextBElemWRTCR();
26
27     // If the cell is equal to Cell2 then return the NextBElemWRTCL
28     } else if (cellIndex == this->getCellLeftIndex()){
29         return getNextBElemWRTCL();
30
31     // If the cell is equal neither the left cell nor the cell right then
32     // throw an exception
33     } else {
34         std::stringstream error;
35         error << "Neither of cell indexes (" << this->getCellLeftIndex() << ", "
36             << this->getCellRightIndex() << ") matches the given one ("
37             << cellIndex << ")";
38
39         throw std::runtime_error(error.str());
40     }
41 }

```


Code listing 2.9 Vertex class is a static vector with defined vector operations. One of the purposes of this class is to be inherited by MeshElement of dimension 0. The listing also presents an example of implementation of addition of two vectors. Using the inlineAddition class, see Code listing 2.10, the loop over vector components is statically unrolled at compile time.

```

1  template <unsigned int Dim, typename Real = double>
2  class Vertex : public std::array<Real, Dim> {
3  public:
4      // Vector subtracion
5      Vertex<Dim, Real> operator-(const Vertex<Dim, Real>&) const;
6      // Vector addition
7      Vertex<Dim, Real> operator+(const Vertex<Dim, Real>&) const;
8      // ... other methods
9  };
10
11 //addition of two vertices
12 template <unsigned int Dim, typename Real>
13 Vertex<Dim, Real> Vertex<Dim, Real>::operator +(const Vertex<Dim, Real>& v) const {
14     Vertex<Dim, Real> res;
15     inlineAddition<Dim, Real>::computation(res.data(), this->data(), v.data());
16     return res;
17 }

```

Code listing 2.10 An example of a an unrolled vector operation. The unrolling is done by class template specialization. The general method adds two elements at the same position and calls addition for the next index. Finally, the specialization for index 0, the function compute only adds the elements at position 0. The main advantage of this approach is the avoidance of loops. Therefore, the compiler can easily optimize such operation. This example is good to understand the point of this trick. It is possible to implement this functionality in more generic way, see Code listing 2.11.

```

1  template <unsigned int N, typename Real>
2  struct inlineAddition{
3      static inline void computation(Real *res, const Real *x, const Real *y){
4          inlineAddition<N-1, Real>::computation(res, x, y);
5          res[N-1] = x[N-1] + y[N-1];
6      }
7      static inline void computation(Real *x, const Real *y){
8          inlineAddition<N-1, Real>::computation(x, y);
9          x[N-1] += y[N-1];
10     }
11 };
12
13 // specialization for argument 0
14 // terminates the recursion
15 template <typename Real>
16 struct inlineAddition<1, Real>{
17     static inline void computation(Real *res, const Real *x, const Real *y){
18         res[0] = x[0] + y[0];
19     }
20     static inline void computation(Real *x, const Real *y){
21         x[0] += y[0];
22     }
23 };

```

Code listing 2.11 An example of a function which automatically performs loop unrolling with respect to vector components and can be applied to any binary operator. In comparison to the implementation in Code listing 2.10, this code is more general, since it allows computation with any binary operator. Furthermore, the implementation utilizes SFINAE [5] to make different implementation of `InlineBinaryProcessor` for different template arguments. The first definition of the function `InlineBinaryProcessor` is applied when the `pos` is lower than `Dim - 1`. If the `pos` is equal to `Dim - 1`, the first definition of `InlineBinaryProcessor` is invalid, therefore the second is used. Note that, when calling this function right one definition must be valid. This way, the recursion can be stopped, when parameter `pos` reaches the value `Dim - 1`. Moreover, the code presents implementation of `operator+` utilizing the `InlineBinaryProcessor`. This concept will be further developed and utilized in Section 4.2.

```

1 // This function template is used when pos < Dim - 1
2 template <unsigned int Dim, typename Real, typename Functor, unsigned int pos = 0>
3 typename std::enable_if<
4     pos < Dim - 1
5 >::type
6 InlineBinaryProcessor(Vertex<Dim, Real>& res,
7     const Vertex<Dim, Real>& op1,
8     const Vertex<Dim, Real>& op2,
9     const Functor& func){
10
11     res[pos] = func(op1[pos], op2[pos]);
12     InlineBinaryProcessor<Dim, Real, Functor, pos+1>(res, op1, op2, func);
13 }
14
15 // This function is valid only when pos == Dim - 1
16 template <unsigned int Dim, typename Real, typename Functor, unsigned int pos = 0>
17 typename std::enable_if<
18     pos == Dim - 1
19 >::type
20 InlineBinaryProcessor(Vertex<Dim, Real>& res,
21     const Vertex<Dim, Real>& op1,
22     const Vertex<Dim, Real>& op2,
23     const Functor& func){
24
25     res[pos] = func(op1[pos], op2[pos]);
26 }
27
28 //addition of two points
29 template <unsigned int Dim, typename Real>
30 Vertex<Dim, Real> Vertex<Dim, Real>::operator +(const Vertex<Dim, Real>& v) const {
31     Vertex<Dim, Real> res;
32     InlineBinaryProcessor(res, *this, v, std::plus<Real>());
33     return res;
34 }

```

Code listing 2.12 Additional properties of cells and faces to simplify the work with the mesh. This class provides integer label `flag` and `center` vertex to its child object. The label can be used to mark boundary cells or faces to determine the applied boundary condition. The center vertex is used to calculate some properties of the mesh, e.g., measures or face normal vectors. Therefore, it simplifies the work with mesh topology.

```

1 template <unsigned int MeshDim, typename Real>
2 class ComputationallySignificantElement
3 {
4     protected:
5         Vertex<MeshDim, Real> center;
6         int flag;
7     public: // ... methods
8 };

```

Code listing 2.13 Definition of general `MeshElement` template class. This definition is intended to be used for elements with dimension between 1 and $d_{\mathcal{T}}$. These elements have to store all references to their sub-elements. The references are stored in special container named `SubelementContainer`, that is, `std::array` for `Reserve` greater than 0 and `std::vector` for `Reserve` equal to 0, see Code listings 2.14 and 2.15. This concept enables to specify the reserved number of references which are directly embedded into the structure, and therefore the optimizes the memory location. However, it is still possible to use the dynamic allocation in case, that the user do not know the loaded mesh parameters. Moreover, using the conditional inheriting, the class also inherits qualities `CellBoundaryConnection` and `ComputationallySignificantElement`, when the dimension of the `MeshElement` is $d_{\mathcal{T}} - 1$. Otherwise, empty structures are inherited, which resolves in omitting the inheritance.

```

1  template <unsigned int MeshDim, unsigned int ElementDim,
2          typename IndexType, typename Real,
3          unsigned int Reserve = 0>
4  class MeshElement : public MeshElementBase<IndexType>,
5                    public std::conditional<ElementDim == MeshDim - 1,
6  CellBoundaryConnection<IndexType>, emptyStruct>::type,
7                    public std::conditional<ElementDim == MeshDim - 1,
8  ComputationallySignificantElement<MeshDim, Real>, emptyStruct2>::type{
9      SubelementContainer<IndexType, Reserve> Subelements;
10 public: ...
11 };

```

We start with the general definition of the `MeshElement` class template and continue with its specializations. To declare the `MeshElement` class, the template arguments are the following:

1. `MeshDim`: the dimension of the mesh,
2. `ElementDim`: the dimension of the particular element,
3. `IndexType`: type of reference,
4. `Real`: type of coordinates,
5. `Reserve`: prescribed number of references.

The general `MeshElement` class is constructed as an element of dimension between 1 and $d_{\mathcal{T}}$, which has a prescribed number of sub-elements. According to the scheme presented in Figures 2.1 and 1.16, it is the connection highlighted in blue. The definition of general `MeshElement` class template is in Code listing 2.13. Moreover, in the case of element dimension $d_{\mathcal{T}} - 1$, the classes `CellConnectingElement` and `ComputationallySignificantElement` are inherited using conditional inheritance. This is achieved by utilizing the `std::conditional` class template which returns the particular type or `emptyStruct` which is equivalent to omitting the inheritance. For example, this structure is applied for construction of the face structure in 3D. Moreover, static allocation of references to sub-elements for positive `Reserve` and dynamic allocation for zero `Reserve` are required according to the point 4 of the list of the mesh element properties in Section 2.2.1. This is achieved in the class template `SubelementContainer`, see Code listings 2.14 and 2.15. From the implementation point of view, `SubelementContainer` is generally `std::array<IndexType, Reserve>`, i.e., all reserved references are embedded into the class at compile time. Specially for zero `Reserve`, it inherits `std::vector<IndexType>`, which is a dynamic container. In both cases, `SubelementContainer` provides a unified interface including the `push_back` method, which is usually available in dynamic data structures only. In the general case when the `Reserve` is to be exceeded by the `push_back` method, a runtime exception is thrown. Finally, to be able to iterate over the `SubelementContainer` correctly, the methods `end` and `cend` must be rewritten to return pointer after last valid element of the container. Otherwise, the whole array would be iterated. See declaration of those methods in Code listing 2.14.

Code listing 2.14 Definition of the `SubelementContainer` template class. This general definition is applied in the case of positive reserve, otherwise a specialization for zero reserve is utilized, see Code listing 2.15. The purpose of this container is to behave as a dynamically allocated vector. Therefore, it has its length and method `push_back`. This way the work with both containers is the same. The only difference is when `push_back` method would overgrow the reserved number of elements, then an exception is thrown. Furthermore, when using iterators on this class, it is necessary to stop the iteration in time. Otherwise, the whole `std::array` would be iterated. This was achieved by redeclaring methods `end` and `cend` to return pointer right after the last valid element.

```

1  template <typename IndexType, unsigned int Reserve>
2  class SubelementContainer : public std::array<IndexType, Reserve>{
3      unsigned int numberOfElements = 0;
4  public:
5      unsigned int getNumberOfSubElements() const {
6          return numberOfElements;
7      }
8
9      unsigned int size() const {
10         return numberOfElements;
11     }
12
13     unsigned int reserve() const {
14         return Reserve;
15     }
16
17     void addSubelement(IndexType index) {
18         if (numberOfElements < Reserve){
19             this->at(numberOfElements) = index;
20             numberOfElements++;
21         } else {
22             throw(std::runtime_error(
23                 "number_of_edges_overgrew_the_number_of_reserved_indexes(" +
24                 std::to_string(Reserve) + ").")
25             );
26         }
27     }
28
29     void push_back(IndexType index) {
30         addSubelement(index);
31     }
32
33     void removeSubelement(unsigned char atIndex){
34         if (atIndex < numberOfElements){
35             for(unsigned char i = atIndex; i < numberOfElements - 1; i++){
36                 this->at(i) = this->at(i+1);
37             }
38             this->at(numberOfElements) = {INVALID_INDEX(IndexType), false};
39             numberOfElements--;
40         } else {
41             throw(std::runtime_error(
42                 "removing_index" + std::to_string(atIndex) +
43                 "is_greather_than_number_of_subelements" +
44                 std::to_string(numberOfElements)+ ".")
45             );
46         }
47     }
48
49     typename std::array<IndexType, Reserve>::iterator end() {
50         return this->begin() + getNumberOfSubElements();
51     }
52
53     typename std::array<IndexType, Reserve>::const_iterator cend() const {
54         return this->cbegin() + getNumberOfSubElements();
55     }
56 };

```

Code listing 2.15 Specialization of `SubelementContainer` class for template argument `Reserve` equal to 0. Unlike to generic definition inheriting `std::array`, shown in Code listing 2.14, this class inherits `std::vector` which is a dynamically allocated container. The methods are only unifying the interface with the generic `SubelementContainer` template.

```

1  template<typename IndexType>
2  class SubelementContainer<IndexType, 0> : public std::vector<IndexType> {
3  public:
4      IndexType getNumberOfSubElements() {
5          return this->size();
6      }
7
8      void addSubelement(IndexType index) {
9          this->push_back(IndexType{index});
10     }
11
12     void removeSubelement(unsigned char atIndex){
13         this->erase(atIndex);
14     }
15 };

```

Code listing 2.16 Specialization for `MeshElement` with dimension 0. The element of 0 dimension is a vertex. Therefore, in this case the element only inherits properties of `MeshElementBase` and `Vertex`, discussed in Code listings 2.6, 2.9.

```

1  template <unsigned int MeshDim, typename IndexType,
2           typename Real, unsigned int Reserve>
3  class MeshElement<MeshDim, 0, IndexType, Real, Reserve>
4      : public MeshElementBase<IndexType>,
5        public Vertex<MeshDim, Real>{
6  public: ...
7  };

```

Code listing 2.17 `MeshElement` specialization for element dimension 1, i.e., edge. The edge is defined by two vertices, therefore the structure have two references to vertices named *A* and *B*. Similarly to the definition of general element, if the mesh dimension is 2, then edge inherits `CellBoundaryConnection` and `ComputationallySignificantElement`.

```

1  template <unsigned int MeshDim,typename IndexType, typename Real, unsigned int Reserve>
2  class MeshElement<MeshDim, 1, IndexType, Real, Reserve>
3      : public MeshElementBase<IndexType>,
4        public std::conditional<MeshDim == 2,
5          CellBoundaryConnection<IndexType>, emptyStruct>::type,
6        public std::conditional<MeshDim == 2,
7          ComputationallySignificantElement<MeshDim, Real>, emptyStruct2>::type{
8  public:
9      IndexType vertexAIndex;
10     IndexType vertexBIndex;
11  public:
12     //... methods
13 };

```

Code listing 2.18 `MeshElement` specialization for element dimension identical to the grid dimension, i.e., cell. The cell element has one reference to one of its boundary faces, `boundaryElementIndex`. Additionally, cell is a computation significant element, therefore it inherits the corresponding property.

```

1  template <unsigned int MeshDim, typename IndexType, typename Real, unsigned int Reserve>
2  class MeshElement<MeshDim, MeshDim, IndexType, Real, Reserve>
3      : public MeshElementBase<IndexType>,
4        public ComputationallySignificantElement<MeshDim, Real>{
5
6      IndexType boundaryElementIndex;
7  public:
8      //...methods
9  };

```

MeshElement Vertex Specialization The first specialization of `MeshElement` is for zero element dimension. This element is actually a vertex in space. Therefore, the previously prepared class `Vertex` can be used with advantage. In this case the `MeshElement` is created by inheritance of `MeshElementBase` and `Vertex`. See Code listing 2.16.

MeshElement Edge Specialization Next `MeshElement` specialization is for an edge, i.e., element with dimension 1. The edge element is defined as a line between two vertices, therefore the edge contains two references to vertices, `vertexAIndex` and `vertexBIndex`. Because it is a mesh element, it inherits `MeshElementBase`. Moreover, in the case of a 2D mesh, the one dimensional `MeshElement` is also the cell connecting element and computationally significant element. Using the same trick of conditional inheritance as in the case of a generic element, the edge `MeshElement` is defined as shown in Code listing 2.17.

MeshElement Cell Specialization The last remaining specialization is for a cell element. It is the `MeshElement` with dimension $d_{\mathcal{T}}$. The construction of the cell element structure is trivial, as it is always computationally significant element and it has one reference to a boundary element. Therefore, this specialization of `MeshElement` class template is created by inheritance of `MeshElementBase` and `ComputationallySignificantElement`, see Code listing 2.18.

2.2.4 Construction of the `MeshElements` structure

In the previous Section 2.2.3 the class template `MeshElement` was thoroughly described. The main advantage of the template class `MeshElement` is its possibility to create the data structure of any mesh element by specifying the template arguments. Thus, the `MeshElement` class is suitable for construction of the most important structure `MeshElements`, which is described in this section.

As shown in Code listing 2.1, it is possible to define a mesh in a general $d_{\mathcal{T}}$ -dimensional space. It is therefore necessary to determine the reserve size for mesh elements of dimension ranging from 2 to $d_{\mathcal{T}} - 1$. Because the mesh dimension is general, the `Reserves` template parameter must be variadic. Specifications of reserves sizes for all relevant dimensions from a parameter pack that has to be processed at compile time. To solve this problem, we utilized the class `std::array` which has the `constexpr` constructor and function `std::get` to obtain a `constexpr` element of the array, see Code listing 2.19. The only `MeshElement` structures which the reserve is applied to are the elements with dimension from 2 to $d_{\mathcal{T}} - 1$. The reserve sizes are specified in the template argument as a comma-separated list in descending order in terms of element dimension (see Code listing 2.1). That is, the first value is for the faces and the last is for the elements with dimension 2. Finally, if the reserve is not set by the user, then the default value is 0, which resolves in dynamic allocation, as discussed in the implementation of `SubelementContainer` in

Code listings 2.14, 2.15. The implementation of reserve determination is presented at lines 5–28 in Code listing 2.19.

The next important step of constructing `MeshElements` is template using-declaration of `ElementType` at lines 35 to 37 in Code listing 2.19. The benefit of the construction of a mechanism returning the reserved number of sub-elements according to the element dimension is now obvious. This step is important because now the type of `MeshElement` of any dimension can be obtained by expression `ElementType<dimension>`.

Next, as we already mentioned, the mesh is a group of `std::vector` classes of `MeshElement` structures with different dimensions. In order to construct the data structure of `MeshElements`, a processor template class `_MeshElements` is used, which is a private class of `MeshElements`. The `_MeshElements` class utilizes an approach similar to implementation of `std::tuple`. It inherits itself with decreased dimension. This way it embeds all the vectors `_data` of different types (`ElementType`) into itself. Finally, the recursion stops with specialization for dimension 0, because then it does not further inherit another `_MeshElements`. See lines 45–53 in Code listing 2.19.

As we prepared the tuple like structure `_MeshElements` containing all the vectors of elements, it only remains to create an instance of `_MeshElements<Dimension> innerElements`, which is ready to represent any unstructured mesh. The definition is at the line 57 of Code listing 2.19.

Now the elements are ready to be used, but we have to approach them from the outside of the class. For this purpose, there is a template method `getElements`. It has one template argument the dimension which specifies the vector to be returned. From the implementation of this method, it could be seen (lines 68–73 of Code listing 2.19) how to extract an attribute from the `_MeshElements` structure according to the dimension.

Furthermore, let us note how the elements are referring to each other, because it is an important part of the architecture. There are two possible solutions:

1. mesh elements have pointers to other elements,
2. mesh elements have a position of an element in a container.

The advantage of the first approach is that it does not require the structure `MeshElements` to approach the connected elements. The elements can be approached directly from other elements. For example, we have a cell and it is possible to approach its boundary face. As is visible from the definition of `MeshElement` class, we decided to use the second solution. The reason is to enable growing of the mesh. When the mesh is changed, e.g., adapted, the vectors could be reallocated and suddenly all pointers would be invalid. On the other hand, the indexes still remain valid. In the case of insertion of new elements into the vector, it is necessary to map old indexes to new indexes. But it is not difficult to create such mapping during the insertion. The problem of approaching the elements in the mesh and mesh iteration is discussed in Section 2.4.1.

In Code listing 2.19, there are shown two more qualities which will be discussed in further sections, `boundaryCells` (Section 2.2.5) and `meshSignature` (Section 2.5.3).

2.2.5 Mesh Boundary

After the structure storing the unstructured mesh is complete, it is important to discuss, what is outside the mesh, i.e., the representation of the domain boundary [28, 17]. In `GTMesh`, the boundary of the tessellated domain is enveloped by virtual cells. Those cells are called boundary cells, see the line 58 in Code listing 2.19. In `OpenFOAM` [17], for example, the mesh boundary can contain any mesh elements, i.e., it is another mesh connected to the inner one. In `GTMesh`, this can be achieved by changing the definition of `std::vector<Cell>` to `_MeshElements<Dimension>`, see Code listing 2.2.5. So far it was not necessary to develop the `GTMesh` with ability to have boundary mesh. There are several reasons why to store the boundary cells separately. Usually the work with the mesh differs at boundary, e.g., treatment

Code listing 2.19 The `MeshElements` class is the class which has containers for all mesh elements. The container for the elements is `innerElements` and is of a type `_MeshElements`. The `_MeshElements` is a class which inherits itself with different dimension parameter, thanks to that it gathers the vectors of `MeshElement` of different dimensions named `elements` in itself. The `_MeshElements` class utilizes the template using declaration of `ElementType`, which simplifies the definitions of `MeshElement` structures. Finally, the `ElementType` utilizes the function `reserve`, which returns a preset reserve from the parameter pack `Reserve` for the given dimension. If the reserve is not set, it returns 0. A detailed description is in Section 2.2.4.

```

1  template <unsigned int Dimension, typename IndexType, typename Real, unsigned int ...Reserve>
2  struct MeshElements{
3  private:
4
5      template<unsigned int dim, typename Void = void>
6      struct _Reserve{
7
8          static unsigned int constexpr value = std::get<Dimension - dim - 1>
9              (std::array<unsigned int, sizeof... (Reserve)>{Reserve...});
10
11     };
12
13     template<unsigned int dim>
14     struct _Reserve<dim,
15         typename std::enable_if<
16             dim == Dimension || dim == 1 || dim == 0 ||
17             (Dimension - dim > sizeof...(Reserve))
18         >::type>{
19
20         static unsigned int constexpr value = 0;
21     };
22 public:
23
24     template<unsigned int dim>
25     static unsigned int constexpr reserve() {
26         return _Reserve<dim>::value;
27     }
28
29     using Vertex = MeshElement<Dimension, 0, IndexType, Real, 0>;
30     using Edge = MeshElement<Dimension, 1, IndexType, Real, 0>;
31     using Face = MeshElement<Dimension, Dimension - 1, IndexType, Real, reserve<Dimension - 1>()>;
32     using Cell = MeshElement<Dimension, Dimension, IndexType, Real, 0>;
33
34     template<unsigned int ElementDimension>
35     using ElementType =
36         MeshElement<Dimension, ElementDimension, IndexType, Real, reserve<ElementDimension>()>;
37
38     static unsigned int constexpr meshDimension() {
39         return Dimension;
40     }
41
42 private:
43
44     template <unsigned int ElemDim = Dimension, typename Dummy = void>
45     struct _MeshElements : public _MeshElements<ElemDim - 1, Dummy>{
46         std::vector<ElementType<ElemDim>> elements;
47     };
48
49     template <typename Dummy>
50     struct _MeshElements<0, Dummy>{
51         std::vector<ElementType<0>> elements;
52     };
53
54 private:
55
56     _MeshElements<Dimension> innerElements;
57     std::vector<Cell> boundaryCells;
58
59     /**
60     * @brief Hash signature of the mesh elements.
61     * Use to detect changes in mesh.
62     */
63     size_t meshSignature;
64
65 public:
66
67     template<unsigned int dim>
68     std::vector<ElementType<dim>>& getElements(){
69         static_assert (Dimension >= dim,
70             "In GetElements template parameter dim must be less or equal to Dimension.");
71         return innerElements._MeshElements<dim>::elements;
72     }
73     // ...
74 };
75

```

Code listing 2.20 An example how to change the boundary elements from cells to a whole mesh.

```

1  _MeshElements<Dimension> innerElements;
2  // MeshElements prepared to envelop the inner elements
3  // std::vector<Cell> boundaryCells;
4  _MeshElements<Dimension> boundaryElements;

```

Code listing 2.21 An example how to make, extract, and detect a boundary element by manipulating the highest bit of its index.

```

1  #include <limits>
2  #define BOUNDARY_INDEX(indexType) \
3      (static_cast<indexType>(1) << (std::numeric_limits<indexType>::digits - 1))
4
5  #define EXTRACTING_INDEX(indexType) (static_cast<indexType>(~BOUNDARY_INDEX(indexType)))
6
7  template <typename IndexType>
8  bool isBoundaryIndex(const IndexType& index){
9      return (BOUNDARY_INDEX(IndexType) & index) == BOUNDARY_INDEX(IndexType);
10 }
11
12
13 template <typename IndexType>
14 IndexType makeBoundaryIndex(const IndexType& index){
15     return (BOUNDARY_INDEX(IndexType) | index);
16 }
17
18
19 template <typename IndexType>
20 IndexType extractBoundaryIndex(const IndexType& index){
21     return (EXTRACTING_INDEX(IndexType) & index);
22 }

```

of boundary conditions, mesh import or export. Therefore, it is necessary to quickly recognize the mesh boundary, or to confidently work only with the inner elements. Another reason is that there are no invalid references in the inner mesh elements. Moreover, the outer cells does not have to be $d_{\mathcal{T}}$ dimensional object, i.e., it may coincide with the face.

Now let us present, how the boundary cells are connected to the mesh and how the references to them are distinguished from the references into the inner mesh. As described in Section 2.2.4, the elements refer to each other by the position in the respective array. The problem is how to refer to a boundary element which is stored in different container. It would be possible to have this information in the flag, but the flag is designed to be used by the user. Therefore, to solve this problem, a special modification of the reference is used. The modification consists in setting the highest bit of the reference to 1, see an example in Code listing 2.21. Finally, this concept allows us to distinguish what element the reference is pointing at. An example of runtime detection whether a cell connected to a face is inner or boundary is shown on trivial example of cell distance calculation in Code listing 2.22.

2.3 Data Associated to the Mesh

Any numerical method for solving a system of PDEs on a domain tessellated by the mesh needs to store data associated with the individual mesh elements. The data can represent the values of the solution, auxiliary pre-calculated space-dependent quantities or implementation-specific storage for intermediate results. The purpose (and hence the type) of data associated to mesh elements is specific to their dimension. For example, finite volume methods (see Section 5.2.2) use solution values at the computationally significant elements (see Section 2.2.2), i.e., cell centers, cell faces, or both. However, additional data storage may be allocated for each vertex to hold the results

Code listing 2.22 An example of getting the correct cell according to its index. This for loop calculates distances between the neighboring cell centers over all faces. If the cell index is a boundary index, then a cell from the array of boundary cells is obtained, else the inner cell is obtained.

```

1 for(auto& face : mesh.getFaces()) {
2     auto& cellLeft = (isBoundaryIndex(face.getCellLeftIndex()) ?
3         mesh.getBoundaryCells().at(extractBoundaryIndex(face.getCellLeftIndex())) :
4         mesh.getCells().at(face.getCellLeftIndex()));
5     auto& cellRight = (isBoundaryIndex(face.getCellRightIndex()) ?
6         mesh.getBoundaryCells().at(extractBoundaryIndex(face.getCellRightIndex())) :
7         mesh.getCells().at(face.getCellRightIndex()));
8
9     distances.at(face) = (cellLeft.getCenter() - cellRight.getCenter()).normEuclid();
10 }

```

of intermediate calculations.

2.3.1 Properties of the MeshDataContainer Class

Once the mesh geometry and topology is given, `MeshDataContainer` provides a flexible interface for allocating and accessing the mesh-associated data. In the most generic case, a single instance of `MeshDataContainer` is capable of holding data represented by types T_1, T_2, \dots, T_n associated with all mesh elements of dimensions d_1, d_2, \dots, d_n , respectively. The dimensions specifiers (d_1, d_2, \dots, d_n) need not be unique, i.e., there can be more than one data type associated with elements of the given dimension.

Internally, `MeshDataContainer` contains n arrays with interface similar to `std::vector<T1>` ... `std::vector<Tn>`. For each $i \in 1, \dots, n$, the length of the vector is the same as the number of mesh elements of dimension d_i .

The vectors within `MeshDataContainer` can be addressed in two ways:

1. by position i of the dimension d_i within the ordered list (d_1, d_2, \dots, d_n) ,
2. by dimension d . In this case, the i th vector is returned where i is the first integer in the sequence $(1, 2, \dots, n)$ such that $d_i = d$.

In addition, the data within `MeshDataContainer` can also be indexed directly by the instances of `MeshElement`, i.e., the `MeshDataContainer` provides a subscript operator for instances of the `MeshElement` class (see Code listing 2.28 with application example in Code listing 2.4). This establishes a mapping between mesh elements and data instances. The data vector in `MeshDataContainer` is given by the dimension of the mesh element (by using the rule explained above). The component of the vector is given by `ElementIndex`, which is present in each `MeshElement` thanks to its base class `MeshElementBase` (see Code listing 2.6).

2.3.2 Construction of MeshDataContainer

In this section, the construction of `MeshDataContainer` as a plain container is firstly described. Then, the methods for allocation and obtaining data are presented.

The template arguments of `MeshDataContainer` are typename `DataType` and parameter pack `unsigned int Dimensions` (i.e., the sequence d_1, d_2, \dots, d_n described above). As a data container, we do not use `std::vector` directly. Instead, the `DataContainer` structure is used, which inherits `std::vector` and additionally has the information about the mapped dimension. The `DataType` can be a simple data type or list of types (i.e., the sequence T_1, T_2, \dots, T_n) passed using `std::tuple`. In the case of a simple type, then `MeshDataContainer` creates `DataContainer` of the given type `DataType` for all dimensions. In the case of `std::tuple<T1, T2, ..., Tn>` as the `DataType`, the tuple serves as a container of the types and the contained types are then used

Code listing 2.23 Presentation of the usage of the `MeshDataContainer` in the most general case. The definition at line 2 defines a container allocating 4 different arrays according to the four numbers passed as comma-separated list. The values in the list specify the dimensions of the associated mesh elements, i.e., the first and last arrays maps the data to mesh elements with dimension 3 and the second and the third arrays maps the data to dimension 2. As the example presents, the list of the dimensions might contain duplicities. The datatypes of the respective arrays are specified by the corresponding type in the `std::tuple` passed as the first parameter, i.e., the first array contains `char`, the second `int` etc. Lines 4-14 explain how data arrays are addressed by the position in the list of dimensions by the `getDataByPos` member function. At lines 16-20, `getDataByDim` returns the first array associated with mesh elements of the requested dimension. Therefore, it is not possible to reach the arrays at positions 2 and 3 by the member function `getDataByDim`. The `std::vector` is quoted in the comments because `MeshDataContainer` utilizes its own similar container, see the further description of the `MeshDataContainer` class in Sections 2.3.1, 2.3.2.

```

1 // Presentation of MeshDataContainer properties
2 MeshDataContainer<std::tuple<char, int, double, float>, 3,2,2,3> meshData;
3
4 // Returns the container "std::vector<char>" mapped to dimension 3
5 meshData.getDataByPos<0>();
6
7 // Returns the container "std::vector<int>" mapped to dimension 2
8 meshData.getDataByPos<1>();
9
10 // Returns the container "std::vector<double>" mapped to dimension 2
11 meshData.getDataByPos<2>();
12
13 // Returns the container "std::vector<float>" mapped to dimension 3
14 meshData.getDataByPos<3>();
15
16 // Get the first data mapped to the 3rd dimension
17 meshData.getDataByDim<3>(); // Returns getDataByPos<0>()
18
19 // Get the first data mapped to the 2nd dimension
20 meshData.getDataByDim<2>(); // Returns getDataByPos<1>()

```

according to their position in the tuple, see Code listing 2.25. The type matches the dimension at the same position. The definition of `DataContainer` and basic definition of `MeshDataContainer` for simple `DataType` and specialization for `std::tuple` are shown in Code listing 2.25.

To be able to work with the container, member functions for accessing the data must be developed. In the previous Section 2.3.1, we specified two ways how to reach the allocated data. The first and easier is to use the position index, realized by `getDataByPos` member function template with the template argument `pos`, see Code listing 2.26. The other is to access the `DataContainer` by the mapped dimension. In this case, it is necessary to find the position of the requested dimension in the parameter pack `Dimensions` at the compile time.

This problem is solved by the `dimensionIndex` function which utilizes the template structure `DimensionPos`. `DimensionPos` is constructed such that it inherits itself with a position increased by 1. If the dimension at the current position matches the given dimension, a specialization of `DimensionPos` is utilized. This specialization has a member function `res` returning the current position. Thanks to this construction, the member function `res` can be called from the generic instance of `DimensionPos`. If the requested dimension is not in the parameter pack `Dimensions`, there is no inherited specialization in `DimensionPos`. Therefore, it does not have the method `res` and the compilation fails. Similarly, position exceeding the size of the parameter pack also causes an error. Finally, the method `getDataByDim` returns `DataContainer` at the position corresponding to the given dimension. The implementation of `dimensionIndex` and `getDataByDim` is in Code listing 2.27.

Using the method `getDataByDim`, it is possible to realize a subscript operator, which has an instance of `MeshElement` as a parameter. The subscript operator returns a reference to the `DataType` instance in the respective `DataContainer`. This construction gives the user a feeling that the data are mapped to the mesh and makes the code more readable. The definition is shown in Code listing 2.28.

Finally, we explain data allocation according to the instance of `MeshElements`, see line 18 in Code listing 2.4. The data allocation is realized by the helper class `Allocator`, because this problem requires a compile-time loop over all instances of `DataContainer` in `MeshDataContainer`. This class has member functions that accept `MeshDataContainer` and `MeshElements` and set the size of `DataContainer` according to the number of mesh elements in the given dimension. The simplest implementation of data allocation of `MeshDataContainer` is shown in Code listing 2.29. Finally, the data allocation is provided by the member function `allocateData`. The same functionality is provided by the constructor of the `MeshDataContainer` class.

The function `allocateData` presented above allocates uninitialized blocks of memory, however the `MeshDataContainer` provides also data allocation with given initial value, similarly to `resize` member function of `std::vector`. This functionality is provided by for both single data type or a tuple of types. In the case of a single data type, one initial value is expected and it is applied initialization of all containers. Otherwise, in the case of the sequence of types T_1, T_2, \dots, T_n , the function `allocateData` expects up to n initial values corresponding to the given types. The values are used to initialize the `DataContainer` at corresponding position. If the number of initial values is lower than n , the `DataContainer` structures without prescribed initial value are allocated uninitialized.

2.3.3 Generation of MeshDataContainer Using `std::integer_sequence`

The next problem we have arrived at can be demonstrated on the following example. Consider a function that calculates the elements properties across multiple dimensions at once, e.g., measures of elements, see Section 2.4.6. Such function has to return all measures of all elements except the vertices, because measure of vertices is irrelevant. In 2D, the resulting container would be `MeshDataContainer<double, 1, 2>`, whereas in 3D it would be `MeshDataContainer<double, 1, 2, 3>`. The problem is, how to automatically specify the sequence of the dimensions the data are mapped to.

Code listing 2.24 The `DataContainer` is a class designed to be used by `MeshDataContainer`, which maps data to mesh elements. Therefore, the `DataContainer` inherits a vector and additionally has an attribute of the mapped dimension, i.e., the dimension of elements the data is mapped to.

```
1 template<typename DataType, unsigned int MappedDimension = 0>
2 struct DataContainer : public std::vector<DataType> {
3     using type = DataType;
4
5
6     static constexpr unsigned int getMappedDimension() {
7         return MappedDimension;
8     }
9 };
```

There is a solution in the standard library named `std::integer_sequence`. This object carries a sequence of integers in its template parameters. This is good structure to be used, but the problem changed to how to automatically generate such `integer_sequence`. In the standard template library, there is an alias named `std::make_integer_sequence`, which generates a sequence starting with 0 going to the given number minus one [4]. For the purposes mentioned above, we need to specify both the first and the last value of the sequence to be generated. By engaging a similar concept to the standard one the `make_custom_integer_sequence_t` was developed, see Code listing 2.31. This alias uses `MakeCustomIntegerSequence` to generate the requested `std::integer_sequence`.

When the sequence is prepared, it still remains to use it to create a `MeshDataContainer`. The easiest way is to make a template class with two type parameters `DataType` and `Sequence`. The interesting parameter is the `typename Sequence` here, because it is designed for an `integer_sequence` to be passed through. Finally, the specialization of `MakeMeshDataContainer` for `Sequence` of type `std::integer_sequence` of `unsigned int` reaches the contained sequence of indexes and creates a public alias type of `MeshDataContainer` with the given mapped dimensions. In the end, for easier usage, there is an alias for the type in `MakeMeshDataContainer`, see Code listing 2.32. An example of the utilization of `make_custom_integer_sequence` to define `MeshDataContainer` is in Code listing 2.30.

Code listing 2.25 The generic definition of `MeshDataContainer`. For a single data type, `MeshDataContainer` defines a private member class `_DataContainer`, which gradually inherits itself similarly to `_MeshElements`, see Code listing 2.19. This way, the `DataContainers` are mapped to the given dimensions at compile time. The data are held in the attribute `data` of type `_DataContainer`. In the case of the specialization for `std::tuple`, the only difference is that the types in `_DataContainer` differ according to the given position. The containers are generated using the number of dimensions using keyword `sizeof... (Dimensions)`. The dimension at the particular position is determined by `std::get` and the `constexpr` constructor of `std::array`, see the definition of `dimensionAt` at lines 5 and 30, respectively. Finally, the specialization for `std::tuple` obtains the type at the given position using `std::tuple_element`, see the using-declaration `DataType`.

```

1  template <typename DataType, unsigned int ...Dimensions>
2  struct MeshDataContainer{
3  private:
4      template<unsigned int pos>
5      static constexpr unsigned int dimensionAt(){
6          return std::get<pos>(<
7              std::array<unsigned int, sizeof... (Dimensions)>{Dimensions...}
8          );
9      }
10
11     template<typename _DataType, unsigned int Pos>
12     struct _DataContainer : _DataContainer<_DataType, Pos - 1> {
13         DataContainer<_DataType, dimensionAt<Pos>()> _data;
14     };
15
16     template<typename _DataType>
17     struct _DataContainer<_DataType, 0>{
18         DataContainer<_DataType, dimensionAt<0U>()> _data;
19     };
20
21     _DataContainer<DataType, sizeof... (Dimensions) - 1> data;
22 // ... other members
23 };
24
25 // Specialization for std::tuple as a types container
26 template <typename ...DataTypes, unsigned int ...Dimensions>
27 struct MeshDataContainer<std::tuple<DataTypes...>, Dimensions...>{
28 private:
29     template<unsigned int pos>
30     static constexpr unsigned int dimensionAt(){
31         return std::get<pos>(<
32             std::array<unsigned int, sizeof... (Dimensions)>{Dimensions...}
33         );
34     }
35
36     template<unsigned int pos>
37     using DataType = typename std::tuple_element<pos, std::tuple<DataTypes...>>::type;
38
39     template<unsigned int Pos, typename Dummy = void>
40     struct _DataContainer : _DataContainer<Pos - 1, Dummy>{
41         DataContainer<DataType<Pos>, dimensionAt<Pos>()> _data;
42     };
43
44     template<typename Dummy>
45     struct _DataContainer<0, Dummy>{
46         DataContainer<DataType<0>, dimensionAt<0>()> _data;
47     };
48
49     _DataContainer<sizeof... (Dimensions) - 1> data;
50 // ... other members
51 };

```

Code listing 2.26 The `getDataByPos` member function of `MeshDataContainer` returns `DataContainer` at the requested position. The container is obtained using the construction `data._DataContainer<pos>::_data`, which specifies the member of `_DataContainer` with its fully qualified name, because there is more than one member with the same name `_data` in the `_DataContainer` structure. The definition of `_DataContainer` is in Code listing 2.25.

```
1  template<unsigned int pos>
2  DataContainer<DataType, dimensionAt<pos>()& getDataByPos(){
3      return data._DataContainer<DataType, pos>::_data;
4  }
5
6
7  //tuple specialization
8  template<unsigned int pos>
9  DataContainer<DataType<pos>, dimensionAt<pos>()& getDataByPos(){
10     return data._DataContainer<pos>::_data;
11 }
```

Code listing 2.27 The implementation of the member function of `MeshDataContainer` `getDataByDim`. This function works similarly to `getDataByPos`, but it returns data on the first position with the requested dimension. To detect the position of the dimension, it uses the `constexpr` member function `dimensionIndex`. `dimensionIndex` utilizes the structure template `DimensionPos`, which tests whether the dimension at the given position is the same as requested by inheriting itself with increased position. If the parameter pack `Dimensions` contains the requested dimension, a specialization of `DimensionPos` for equality of current and requested dimensions is utilized at the respective position. The specialization stops the recursive inheritance. Moreover, it provides the `res` member function which returns the found position. The method `res` is defined in the class `DimensionPos<requested_dim,0,dim_at_0>`, see line 18. Otherwise, if the requested dimension is not in `Dimensions`, then the attribute `pos` in `DimensionPos` exceeds the number of elements in the `Dimensions` parameter pack what causes an error.

```

1 // **Inside MeshDataContainer**
2 template<unsigned int dim, unsigned int pos, unsigned int _dim>
3 struct DimensionPos : DimensionPos<
4     dim,
5     pos + 1, std::get<pos + 1>(
6         std::array<unsigned int, sizeof... (Dimensions)>{Dimensions...}
7     )>{};
8
9 // specialization for dim == _dim
10 template<unsigned int dim, unsigned int pos>
11 struct DimensionPos<dim, pos, dim>{
12     static constexpr unsigned int res(){return pos;}
13 };
14
15 public:
16 template<unsigned int dim>
17 static constexpr unsigned int dimensionIndex(){
18     return DimensionPos<
19         dim,
20         0,
21         std::get<0>(std::array<unsigned int, sizeof... (Dimensions)>{Dimensions...})
22     >::res();
23 }
24
25 template<unsigned int dim>
26 DataContainer<DataType, dim>& getDataByDim(){
27     return data._DataContainer<DataType, dimensionIndex<dim>()>::_data;
28 }
29
30 // tuple specialization
31 template<unsigned int dim>
32 DataContainer<std::tuple_element_t<dimensionIndex<dim>(), std::tuple<DataTypes...>, dim>&
33     getDataByDim(){
34     return data._DataContainer<dimensionIndex<dim>()>::_data;
35 }

```

Code listing 2.28 MeshDataContainer provides the subscript operator (`operator[]`) and the `at` member function with an argument of `MeshElement`. The `operator[]` is a function template because of the template argument `element`. All the template arguments of the method are deduced from the `MeshElement` parameter, so the usage of `at` is easy and it is possible to invoke `operator[]` by the usual syntax (i.e., `x[i]`). Both the functions utilizes `getDataByDim` to obtain the corresponding array, then they return the element in the array at position obtained by the `getElementIndex` member function of `element`. See the usage of subscript operator in Code listing 2.4.

```

1  template <unsigned int ElementDim, unsigned int Dimension,
2          typename IndexType, typename Real, unsigned int Reserve>
3  DataType&
4  at(const MeshElement<Dimension, ElementDim, IndexType, Real, Reserve>& element) {
5      return getDataByDim<ElementDim>().at(element.getIndex());
6  }
7
8  template <unsigned int ElementDim, unsigned int Dimension,
9          typename IndexType, typename Real, unsigned int Reserve>
10 DataType&
11 operator[](const MeshElement<Dimension, ElementDim, IndexType, Real, Reserve>& element) {
12     return getDataByDim<ElementDim>()[element.getIndex()];
13 }
14
15
16 // specialization for tuple
17 template <unsigned int ElementDim, unsigned int Dimension,
18         typename IndexType, typename Real, unsigned int Reserve>
19 std::tuple_element_t<dimensionIndex<ElementDim>(), std::tuple<DataTypes...>&
20 at(const MeshElement<Dimension, ElementDim, IndexType, Real, Reserve>& element) {
21     return getDataByDim<ElementDim>().at(element.getIndex());
22 }
23
24 template <unsigned int ElementDim, unsigned int Dimension,
25         typename IndexType, typename Real, unsigned int Reserve>
26 std::tuple_element_t<dimensionIndex<ElementDim>(), std::tuple<DataTypes...>&
27 operator[](const MeshElement<Dimension, ElementDim, IndexType, Real, Reserve>& element) {
28     return getDataByDim<ElementDim>()[element.getIndex()];
29 }

```

Code listing 2.29 The implementation of the simplest way of data allocation within `MeshDataContainer`. The `allocateData` member function accepts parameter of the `MeshElements` type named by `mesh`. The function `allocateData` utilizes a helper class template `Allocator`. The `Allocator` class iterates over all positions of the `MeshDataContainer` and invokes `resize` of the respective `DataContainer`. Every `DataContainer` is resized to the number of elements in `mesh` at the dimension corresponding to the currently processed `DataContainer`. An analogous approach is used in the specialization of `MeshDataContainer` for `std::tuple` of types. Moreover, `MeshDataContainer` provides several overloads of the `allocateData` member function accepting an initial value (values in the case of tuple of types) similarly to the member function `resize` of the `std::vector` class.

```

1 // **Inside MeshDataContainer**
2 template<unsigned int pos, typename dummy = void>
3 struct Allocator{
4
5     template<unsigned int Dimension, typename IndexType, typename Real,
6             unsigned int ...Reserve>
7     static void allocateMemory(
8         MeshDataContainer<DataType, Dimensions...>& parent ,
9         const MeshElements<Dimension, IndexType, Real, Reserve...>& mesh
10    ) {
11        parent.template getDataByPos<pos>().resize(
12            mesh.template getElements<parent.template dimensionAt<pos>()>().size()
13        );
14
15        // next iteration
16        Allocator<pos - 1>::allocateMemory(parent, mesh);
17    }
18 };
19
20 // specialization terminating the cycle
21 template<typename dummy>
22 struct Allocator<0, dummy>{
23
24     template<unsigned int Dimension, typename IndexType, typename Real,
25             unsigned int ...Reserve>
26     static void allocateMemory(
27         MeshDataContainer<DataType, Dimensions...>& parent ,
28         const MeshElements<Dimension, IndexType, Real, Reserve...>& mesh
29    ) {
30        parent.template getDataByPos<0>().resize(
31            mesh.template getElements<parent.template dimensionAt<0>()>().size()
32        );
33    }
34 };
35
36 template <unsigned int Dimension, typename IndexType, typename Real,
37         unsigned int ...Reserve>
38 void allocateData(const MeshElements<Dimension, IndexType, Real, Reserve...>& mesh){
39     Allocator<sizeof... (Dimensions) - 1>::allocateMemory(*this, mesh);
40 }

```

Code listing 2.30 An example of automatic generation of `MeshDataContainer` for the given range of dimensions. The motivation is described at the beginning of Section 2.3.3.

```

1 // equivalent to MeshDataContainer<double, 1, 2, 3>
2 MakeMeshDataContainer_t<
3     double,
4     make_custom_integer_sequence_t<unsigned int, 1, 3>
5     > measures = mesh.calculateMeasures();

```

Code listing 2.31 Generation of a custom `integer_sequence`. The structure `MakeCustomIntegerSequence` creates a sequence from `StartIndex` to `EndIndex` by adding the `Incerement`. This is reached by recursive inheritance with incremented `StartIndex` the `StartIndex` appended to the template parameter `Sequence`. In the case when `StartIndex` is equal to `EndIndex`, a specialization of `MakeCustomIntegerSequence` is applied. The specialization creates an alias of `std::integer_sequence` with the aggregated `Sequence` and additionally, the `EndIndex` is included. Note that `Sequence` always contains `StartIndex` and `EndIndex`. Furthermore, if the `EndIndex` is not reached by adding `Increment` to `StartIndex`, the build fails by exceeding the maximal recursion depth. Finally, a type alias is provided to make the work easier.

```

1  template<typename Type,
2      Type StartIndex,
3      Type EndIndex,
4      int Increment = 1,
5      Type... Sequence>
6  struct MakeCustomIntegerSequence :
7      public MakeCustomIntegerSequence<
8          Type,
9          StartIndex + Increment,
10         EndIndex,
11         Increment,
12         Sequence...,
13         StartIndex
14     > {};
15
16  template <typename Type, Type EndIndex, int Increment, Type... Sequence>
17  struct MakeCustomIntegerSequence<Type, EndIndex, EndIndex, Increment, Sequence...> {
18      using type = std::integer_sequence<Type, Sequence..., EndIndex>;
19  };
20
21  template<typename Type, Type StartIndex, Type EndIndex, int Increment = 1>
22  using make_custom_integer_sequence_t =
23      typename MakeCustomIntegerSequence<Type, StartIndex, EndIndex, Increment>::type;

```

Code listing 2.32 The `MeshDataContainer` is generated using the `MakeMeshDataContainer` structure, which has two template parameters. The first parameter, `DataType` is a name of the type to be stored. This can be either a simple type or tuple of types. The second parameter, `Sequence`, `std::integer_sequence` of `unsigned int` type, which is ensured by the assertion at line 3. If `std::integer_sequence` is passed as `Sequence`, the compiler chooses the provided specialization. Otherwise, the generic template is used where the assertion is designed to always fail. Finally, the specialization obtains the sequence stored in the type information of `std::integer_sequence` thanks to the template parameter pack `Dimensions`. The parameter `Dimensions` is then used to create an alias for `MeshDataContainer`, see line 13.

```

1  template<typename DataType, typename Sequence>
2  struct MakeMeshDataContainer {
3      static_assert(
4          std::is_class<Sequence>::value && !std::is_class<Sequence>::value,
5          "The Sequence parameter in MakeMeshDataContainer"
6          "must be a std::integer_sequence<unsigned int, seq...>"
7          "please notice the type unsigned int"
8          );
9  };
10
11  template<typename Type, unsigned int... Dimensions>
12  struct MakeMeshDataContainer<Type, std::integer_sequence<unsigned int, Dimensions...>>{
13      using type = MeshDataContainer<Type, Dimensions...>;
14  };
15
16  template<typename DataType, typename Sequence>
17  using MakeMeshDataContainer_t = typename MakeMeshDataContainer<DataType, Sequence>::type;

```

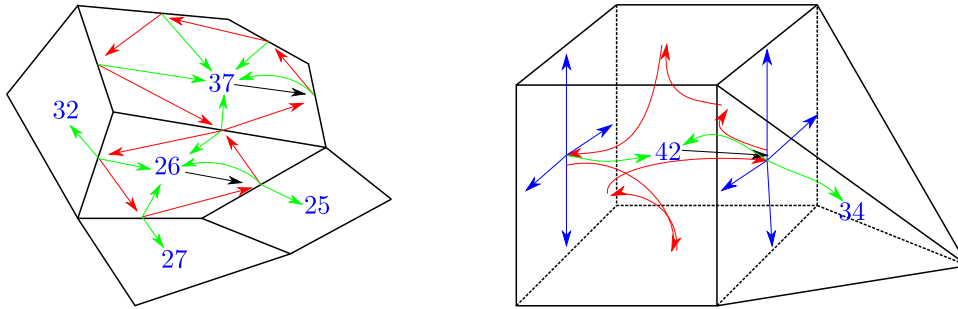


Figure 2.2: A combination of Figure 1.15 and Figure 1.14 reminding of the connections of the mesh elements in 2D (left) and 3D (right) realized in the `MeshElements` structure. The detailed description is in Sections 1.3.3, 2.2.

2.4 Mesh Algorithms

In the previous sections, the mesh construction and data allocation were explained. The aim of this section is to describe the work with the mesh and algorithms that calculate the geometrical properties of mesh elements. Mesh algorithms are implemented as global function templates compatible with the template arguments of `MeshElements`. Template specializations and procedures are implemented in classes, because in C++, it is not possible to partially specialize a function template. It is possible to achieve a similar behavior using SFINAE, however this concept is utilized in another part of the library which will be discussed in Chapter 4. Note that for demonstration purposes, the presented listings may contain a code slightly different from the real implementation.

2.4.1 Element Access and the `MeshApply` Class

The first realized algorithm accesses the elements of a mesh. Its purpose is to iterate over the mesh elements of the given dimension (the “target” dimension) connected to the elements of another dimension (“start” dimension). For example, it visit all vertices of all cells. Moreover the algorithm applies a function on each visited connected mesh element. This is a very important algorithm which will be further employed by other functions. Before we start with the description of the algorithm, let us recall the system of connections in 2D and 3D meshes in Figure 2.2.

In GTMesh, this functionality is provided by the `MeshApply` class template with the `apply` static member function, where the start and target dimensions are set as template parameters of the `MeshApply` class. The requested functionality of `MeshApply` is presented in Code listing 2.33. As shown in the example, the iteration over connected mesh elements requires nested `for` cycles. Therefore, `MeshApply` must de facto encapsulate such `for` cycles and the number of the nested `for` cycles is given by the template setup (start and target dimensions). Additionally, it must apply a given function to each visited connected mesh element. From the example, it is obvious that `MeshApply` provides a more generic and compact means to iterate over the mesh elements. Moreover, the GTMesh concepts are not limited to 3 dimensions. Therefore, `MeshApply` plays a key role in the GTMesh library and makes it possible to create generic functions working in any dimension.

Let us continue with the description of the implementation details of the class template `MeshApply` and its public static method `apply`. Firstly, we have to start with the description of accessing sub-elements of mesh elements which is dependent on the dimension of the mesh element. There are basically 3 different ways of accessing sub-elements according to the element dimension. Note that all of the accesses are presented in the 3D version of the function `countCellsVerticesIndexes` in Code listing 2.33. In the following description, we will refer to the individual lines in Code listing 2.33. The following list introduces the 3 ways of the iteration

Code listing 2.33 Demonstration of iteration over mesh sub-elements. For demonstration purpose, a problem of summing the indexes of vertices is solved. In the `countCellsVerticesIndexes` function, the algorithm specific for mesh dimensions 2 and 3 is realized. This function utilizes the explicitly written loops. Therefore, this function serves as good example of mesh elements access. Moreover, the aim of this example is to show the rigidity of the explicit writing of the loops compared to using the `MeshApply` concept. Note that there are two versions of `countCellsVerticesIndexes` to cover 2D and 3D mesh cases. In contrast to that, the function `countCellsVerticesIndexesGeneric` has a shorter body and is able to handle a mesh of an arbitrary dimension. It would not be possible to create such generic code by using the loops explicitly.

```

1 // Function calculating sum of vertex indexes connected to a cell in 3D
2 template<typename IndexType, typename Real, unsigned int ...Reserve>
3 MeshDataContainer<IndexType,3>
4 countCellsVerticesIndexes(const MeshElements<3, IndexType, Real, Reserve...>& mesh){
5     // Prepare container for the result
6     // Resize to the mesh dimensions and initialize with 0
7     MeshDataContainer<IndexType, 3> result(mesh, 0);
8
9     // Loop over cell elements
10    for(IndexType cellIndex = 0; cellIndex < mesh.getCells().size(); cellIndex++){
11        // Obtain the boundary element index (face)
12        IndexType bElemIndex = mesh.getCells()[cellIndex].getBoundaryElementIndex();
13
14        // Repeat until the first approached face element is reached
15        do {
16            // Loop over the edges of the face
17            for(IndexType edgeIndex : mesh.getFaces()[bElemIndex].getSubelements()){
18                // Edge has two vertices
19                result.template getDataByDim<3>()[cellIndex] += mesh.getEdges()[edgeIndex].getVertexAIndex();
20                result.template getDataByDim<3>()[cellIndex] += mesh.getEdges()[edgeIndex].getVertexBIndex();
21            }
22            // Move to the next boundary element
23            bElemIndex = mesh.getFaces()[bElemIndex].getNextBElem(cellIndex);
24
25        } while (bElemIndex != mesh.getCells()[cellIndex].getBoundaryElementIndex());
26    }
27    return result;
28 }
29
30 // Function calculating sum of vertex indexes connected to a cell in 2D
31 template<typename IndexType, typename Real, unsigned int ...Reserve>
32 MeshDataContainer<IndexType,2>
33 countCellsVerticesIndexes(const MeshElements<2, IndexType, Real, Reserve...>& mesh){
34     // Prepare container for the result
35     // Resize to the mesh dimensions and initialize with 0
36     MeshDataContainer<IndexType, 2> result(mesh, 0);
37
38     // Loop over cell elements
39    for(IndexType cellIndex = 0; cellIndex < mesh.getCells().size(); cellIndex++){
40        // Obtain the boundary element index (face=edge)
41        IndexType bElemIndex = mesh.getCells()[cellIndex].getBoundaryElementIndex();
42
43        // Repeat until the first approached element is reached
44        do {
45            // Because the boundary element is an edge, it has two vertices
46            result.template getDataByDim<2>()[cellIndex] += mesh.getEdges()[bElemIndex].getVertexAIndex();
47            result.template getDataByDim<2>()[cellIndex] += mesh.getEdges()[bElemIndex].getVertexBIndex();
48
49            // Move to the next boundary element
50            bElemIndex = mesh.getEdges()[bElemIndex].getNextBElem(cellIndex);
51
52        } while (bElemIndex != mesh.getCells()[cellIndex].getBoundaryElementIndex());
53    }
54    return result;
55 }
56
57 // Function calculating sum of vertex indexes connected to a cell in generic dimension
58 template<unsigned int MeshDim, typename IndexType, typename Real, unsigned int ...Reserve>
59 MeshDataContainer<IndexType, MeshDim>
60 countCellsVerticesIndexesGeneric(const MeshElements<MeshDim, IndexType, Real, Reserve...>& mesh){
61     // Prepare container for the result
62     // Resize to the mesh dimensions and initialize with 0
63     MeshDataContainer<IndexType, MeshDim> result(mesh, 0);
64
65     // Prepare the function to be applied in the loop
66     auto countLambda = [&result](IndexType cellIndex, IndexType vertIndex){
67         // Add the index of vertex at the position
68         result.template getDataByDim<MeshDim>()[cellIndex] += vertIndex;
69     };
70
71     // Loop over cells vertices using the MeshApply concept
72     MeshApply<MeshDim, 0>::apply(mesh, countLambda);
73
74     return result;
75 }

```

step by step.

1. The sub-elements of cells, i.e., faces, are chained as linked list. The algorithm of looping the cell boundary is the following. Firstly, get the index of any boundary element of the cell shown at line 12. Then, start an iteration over the faces. The index of the next face is obtained from the currently visited face as the next face with respect to the cell, see line 23. This iteration ends when a face with index with starting index is reached, see the condition at line 25.
2. The generic elements with dimension between 1 and $d_{\mathcal{T}}$ (described in Section 1.3.3) have a container of sub-elements indexes. Therefore, to obtain the indexes of sub-elements, it is sufficient to loop over the container in the `for` loop starting at line 17. In 3D, the only dimension between 1 and $d_{\mathcal{T}}$ is dimension 2, thence the procedure is applied to faces only.
3. In a generic case, by applying the second way of iteration, we finally reach edges. The edge has two indexes to vertices A and B . Hence, the indexes of the connected vertices are accessed directly by the `getVertexAIndex` and `getVertexBIndex` member functions at lines 19 and 20.

Since the functionality of `MeshApply` was presented in detail, let us move forward to the implementation. As was already mentioned, the algorithm is implemented in the `apply` member function of `MeshApply`. There are two overloads of the function `apply`. The first one accepts two parameters (a mesh and a function). It traverses all elements of the start dimension and applies the given function to all elements of the target dimension connected to them. The second one has one more parameter (element index) and it invokes the given function on the elements connected to the specified mesh element only.

In order to make the implementation simpler and more compact, the `apply` function internally utilizes the `run` member function of the `MeshRun` class template. The `MeshRun` class provides the iteration over the sub-elements of the target dimension of a single mesh element of the start dimension and application of the given function to them. According to the chosen mesh structure, the iteration is only possible to be made from a higher dimension to a lower dimension. Luckily, there is also a simple way how to simulate the iteration in the opposite direction (iteration over super-elements), although it is not supported by the structure of `MeshElements`. The trick will be described later.

The class `MeshRun` has 6 template parameters:

1. `CurrentDimension` (`unsigned int`), dimension currently iterated over,
2. `StartDimension` (`unsigned int`), dimension at which the recursive iteration started,
3. `TargetDimension` (`unsigned int`), dimension at which the recursion stops and the function is applied,
4. `MeshDimension` (`unsigned int`), dimension of the mesh, necessary for specializations,
5. `End` (`bool`), flag signaling that `CurrentDimension` reached `TargetDimension`,
6. `Descend` (`bool`), flag signaling whether the iteration over sub-elements or super-elements was requested.

The template parameters of the `run` member function template are:

1. `Functor` (`typename`), the type of the given function, i.e., a callable object with two parameters of `IndexType`,
2. The `MeshElements` template attributes (`IndexType` , `Real` , `Reserve`).

Code listing 2.34 The definition of the `MeshRun` class. In the context of the following specializations, this definition is to be applied to iterate over sub-elements of a generic element with dimension between 1 and $d_{\mathcal{T}}$. Those are elements with connections highlighted in blue in Figure 1.16. The iteration over sub-elements is realized by iterating over the `subelementContainer` of the given element. The indexes of sub-elements are then passed as `subelementIndex` to the `run` function of the `MeshRun` class which is given the value of `CurrentDimension` of the invoking `MeshRun` object minus one. The recursive call realizing the next iteration is at lines 24-27. The rest of the parameters given to the function `run` are the `mesh`, the index of the origin element `origElementIndex` and the applied function `fun`.

```

1  template <
2      unsigned int CurrentDimension,
3      unsigned int StartDimension,
4      unsigned int TargetDimension,
5      unsigned int MeshDimension,
6      bool End,
7      bool Descend
8  >
9  struct MeshRun {
10
11      template<typename Functor, typename IndexType, typename Real, unsigned int ...Reserve>
12      static void run(const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh,
13                    IndexType origElementIndex, // Index from which the iteration started
14                    IndexType subelementIndex, // Index of currently visited element
15                    Functor fun // Function to be applied when TargetDimension is reached
16                ){
17          // Get the current element to be processed
18          auto& currentElement =
19              mesh.template getElements<CurrentDimension>().at(subelementIndex);
20          // Loop over the subelements
21          for (auto& sube : currentElement.getSubelements()){
22              // Call MeshRun with a lowered dimension
23              // and check wheter the TargetDimesnion is reached
24              MeshRun<
25                  CurrentDimension - 1, StartDimension, TargetDimension,
26                  MeshDimension, TargetDimension == CurrentDimension - 1, Descend
27              >::run(mesh, origElementIndex, sube, fun);
28          }
29      }
30 };

```

The `run` function accepts four parameters:

1. `mesh(MeshElements)`, the mesh to be iterated,
2. `fun (Functor)`, pointer to a callable object accepting two arguments of `IndexType`,
3. `origElementIndex (IndexType)`, the index of the origin mesh element, i.e., the mesh element the recursion started from,
4. `subelementIndex (IndexType)`, index of the currently visited element.

Note that the origin index is to be passed through the iteration among the dimensions. When the `TargetDimension` is reached, the index of the origin and the currently visited elements are passed to the function `fun`.

Let us proceed to the description of the `MeshRun` specializations and their purpose. The class `MeshRun` consists of one general definition and four specializations. According to the 3 different ways of sub-element iteration, the generic definition and the two specializations are devoted to the iteration of the sub-elements. The remaining two specializations are employed to apply the given function.

- The general `MeshRun` class template is to be utilized for iteration over sub-elements of elements with dimension between 1 and $d_{\mathcal{T}}$. For example, this template is employed in the

Code listing 2.35 Specialization of the class MeshRun for parameters CurrentDimension equal to MeshDimension and End false. This specialization is designed to perform the loop over the boundary elements of cells, and recursively call MeshRun with CurrentDimension lowered by one.

```

1 // Specialization of MeshRun for CurrentDimension == MeshDimension
2 // and End == false
3 template <
4     unsigned int StartDimension,
5     unsigned int TargetDimension,
6     unsigned int MeshDimension,
7     bool Descend
8 >
9 struct MeshRun<
10     MeshDimension,
11     StartDimension,
12     TargetDimension,
13     MeshDimension,
14     false,
15     Descend> {
16
17     template<typename Functor, typename IndexType, typename Real, unsigned int ...Reserve>
18     static void run(const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh,
19                   IndexType origElementIndex,
20                   IndexType subelementIndex,
21                   Functor fun) {
22         // Get the cell element
23         auto& cell = mesh.getCells().at(subelementIndex);
24         // Iteration over the cell boundary
25         // Obtain the first boundary element index
26         IndexType tmpFace = cell.getBoundaryElementIndex();
27         do {
28             // Recursive call of MeshRun with a lowered dimension
29             MeshRun<
30                 MeshDimension - 1, StartDimension, TargetDimension,
31                 MeshDimension, TargetDimension == MeshDimension - 1, Descend
32             >::run(mesh, origElementIndex, tmpFace, fun);
33             // Move to the next boundary element
34             tmpFace = mesh.getFaces().at(tmpFace).getNextBElem(subelementIndex);
35             // Repeat until the origin boundary element is reached
36         } while (tmpFace != cell.getBoundaryElementIndex());
37     }
38 }
39 };

```


Code listing 2.36 Specialization of the class `MeshRun` for parameters `CurrentDimension` equal to 1 and `End` false. The function `run` calls the method `run` of class `MeshRun` with `CurrentDimension` set to 0. Note that there is no specialization for zero dimension and `End` false, because vertices do not have any sub-elements. Therefore, the recursion must be terminated in the next step, i.e., `End` is true (`TargetDimension` is equal to 0). Otherwise, the compilation fails.

```

1 // Specialization of MeshRun for CurrentDimension == 1
2 // and End == false
3 template <
4     unsigned int StartDimension,
5     unsigned int TargetDimension,
6     unsigned int MeshDimension,
7     bool Descend
8 >
9 struct MeshRun<1, StartDimension, TargetDimension, MeshDimension, false, Descend> {
10
11     template<typename Functor, typename IndexType, typename Real, unsigned int ...Reserve>
12     static void run(const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh,
13                   IndexType origElementIndex,
14                   IndexType subelementIndex,
15                   Functor fun){
16         // Get the edge element
17         auto& edge = mesh.getEdges().at(subelementIndex);
18         // Recursive call of MeshRun for indexes of vertices A and B
19         MeshRun<
20             0, StartDimension, TargetDimension,
21             MeshDimension, TargetDimension == 0, Descend
22 >::run(mesh, origElementIndex, edge.getVertexAIndex(), fun);
23
24         MeshRun<
25             0, StartDimension, TargetDimension,
26             MeshDimension, TargetDimension == 0, Descend
27 >::run(mesh, origElementIndex, edge.getVertexBIndex(), fun);
28     }
29 };

```

case iteration over sub-elements of faces in 3D mesh. The definition is presented in Code listing 2.34.

- The first specialization is utilized when `CurrentDimension` is equal to `MeshDimension`, see Code listing 2.35. This specialization is applied in the iteration over a cell boundary.
- The last specialization realizing the sub-element iteration is used when `CurrentDimension` is equal to one. This specialization iterates over the two vertices of edges. The implementation is presented in Code listing 2.36.

All the described iteration methods have a common feature. In the body of the cycle, they recursively call the `run` member function of `MeshRun` with decreased `CurrentDimension` by one. Moreover, they check whether the next iterated dimension is the targeted one and pass it as the `End` template parameter. All three variants of `MeshRun` discussed so far are utilized for parameter `End` equal to false. This is done to prioritize the remaining two specializations. The advantage of this concept is discussed in the end of this section.

If the template parameter `End` is set to true, then the remaining two specializations of `MeshRun` are applied regardless of the other template parameters. Both the specializations apply the given function with the origin and connected mesh elements indexes given and terminate the recursion.

Firstly, let us focus on the function passed to `run`, then the way of passing parameters will be discussed. Since the type of the function is a template parameter, the type of the parameter `fun` could be arbitrary, e.g., a function pointer or `double`. Nevertheless, the function expects the `fun` parameter to be a callable object with two parameters of type `IndexType`. Therefore, the type of the function is checked and if the function does not satisfy the condition, the compilation is terminated with an explanatory error message.

To check the type `Functor`, we utilize the `std::function` class and `std::is_assignable` from the `type_traits` header. Then the properties of `Functor` type are checked by checking whether `Functor` is assignable to `std::function<void(IndexType, IndexType)>`, see the `static_assert` at line 15 in Code listing 2.37. Moreover, the utilization of `std::is_assignable` in combination with `std::function` also automatically resolves the implicit conversion of types. For example, any type can be converted to `void`, therefore it accepts a function with any return type (the return value is not used anyway). After the function is checked it only remains to call it.

The function is passed two indexes of the mesh elements. The first is the index of the origin element of dimension `StartDimension` and the second one is the index of the currently visited element of dimension `TargetDimension`, i.e., the connected one.

As we already mentioned, the mesh data structure enables the iteration over sub-elements only. However, we have already suggested that there is a way how to simulate to loop over the super-elements, i.e., when `StartDimension` is lower than `TargetDimension`. Firstly, let us recall that the `MeshRun` is designed for `StartDimension` higher than `TargetDimension` only. Therefore, the `origElementIndex` is always the index of an element of dimension `StartDimension`. Then the iteration in the opposite direction can be simulated thanks to the symmetry of the connections in the mesh discussed in Section 1.3.1. For this purpose, there is the last template parameter named `Descend`. This parameter signalizes whether the iteration was originally requested in the ascending or descending order. If the `Descend` parameter is true, the `origElementIndex` is passed as the first one, and the currently iterated element index is passed as the second. Otherwise, the order of the passed indexes is swapped. See the lines 22 and 43 in Code listing 2.37. In other words, when the ascending order is requested, the currently visited element is passed as the origin one and the one from which the iteration started is passed as the connected one.

Finally, the functionality of `MeshRun` is wrapped into the `apply` member function of the `MeshApply` class. The purpose of the `apply` function is to prepare all the template parameters for the call of the `run` member function of `MeshRun`, because the class `MeshRun` has many helper template parameters, e.g., `CurrentDimension` or `End`. Moreover, the implementation of `MeshRun`

Code listing 2.37 Specialization of MeshRun for parameter End equal to true. In this case, the run function applies the given function fun with parameters origElementIndex and subelementIndex. There are two versions of the function depending on the template parameter Descend. Both of the functions firstly check whether the passed type Functor is a function which accepts two parameters of IndexType. This is done by testing whether the Functor is assignable to std::function<void(IndexType, IndexType)> using std::is_assignable. If the test fails, then an error with detailed description about the expected Functor properties is raised, see the lines 17 and 37. Finally, the function is called with origElementIndex and subelementIndex given. The order of the indexes is given by the parameter Descend. If it is true, then the origElementIndex is passed as the first one and the subelementIndex as the second one. Otherwise, the order is swapped. This way, the application from lower to higher dimension is simulated. The function shall interpret the first as the origin element and the second one as the connected element.

```

1 #define INVALID_FUNCTOR_DESCR \
2 "The Functor fun must be a function with void return type and two arguments of IndexType, \
3 the first is index of StartDimension element and the second is the index of the \
4 TargetDimension element"
5
6 // Specialization of MeshRun for End == true and Descend == true
7 template < unsigned int CurrentDimension, unsigned int StartDimension,
8           unsigned int TargetDimension, unsigned int MeshDimension >
9 struct MeshRun<CurrentDimension, StartDimension, TargetDimension,
10              MeshDimension, true, true> {
11     template<typename Functor, typename IndexType, typename Real, unsigned int ...Reserve>
12     static void run(const MeshElements<MeshDimension, IndexType, Real, Reserve...>& ,
13                   IndexType origElementIndex,
14                   IndexType subelementIndex,
15                   Functor fun){
16         // Test the Functor type whether it is function
17         static_assert(
18             std::is_assignable<std::function<void(IndexType, IndexType)>, Functor>::value,
19             INVALID_FUNCTOR_DESCR);
20         // Call the function fun with parameters origElementIndex as the origin element
21         // index and the subelementIndex as the index of connected element
22         fun(origElementIndex, subelementIndex);
23     }
24 };
25
26 // Specialization of MeshRun for End == true and Descend == false
27 template < unsigned int CurrentDimension, unsigned int StartDimension,
28           unsigned int TargetDimension, unsigned int MeshDimension >
29 struct MeshRun<CurrentDimension, StartDimension, TargetDimension,
30              MeshDimension, true, false>{
31     template<typename Functor, typename IndexType, typename Real, unsigned int ...Reserve>
32     static void run(const MeshElements<MeshDimension, IndexType, Real, Reserve...>& ,
33                   IndexType origElementIndex,
34                   IndexType subelementIndex,
35                   Functor fun){
36         // Test the Functor
37         static_assert(
38             std::is_assignable<std::function<void(IndexType, IndexType)>, Functor>::value,
39             INVALID_FUNCTOR_DESCR);
40         // Because MeshRun is simulating iteration from TargetDimension
41         // to StartDimension, the order of the passed indexes is swapped
42         // and subelementIndex is passed to fun as the origin one
43         fun(subelementIndex, origElementIndex);
44     }
45 };

```

Code listing 2.38 The `MeshApply` class with the `apply` member function serves as an application wrapper of the `MeshRun` class. Basically, the reason is that `MeshRun` has many helper parameters which can be set just with the knowledge of start, target and mesh dimensions. Moreover, the parameter `MeshDimension` can be derived from the given `MeshElements` instance. Furthermore, using `MeshRun` requires the knowledge which of the start and target dimensions is greater, see lines 3 and 5. There are two versions of the `apply` member function. The first applies `MeshRun` to all elements of the higher dimension. The second one has the parameter `elementIndex` which specifies the single element `MeshRun` will be applied to. The second version of the `apply` member function is limited to `StartDimension` greater or equal to `TargetDimension`.

```

1  template <unsigned int StartDimension, unsigned int TargetDimension>
2  struct MeshApply {
3      static constexpr unsigned int hDim =
4          (StartDimension > TargetDimension) ? StartDimension : TargetDimension;
5      static constexpr unsigned int lDim =
6          (StartDimension > TargetDimension) ? TargetDimension : StartDimension;
7
8      template<unsigned int MeshDimension, typename Functor,
9              typename IndexType, typename Real, unsigned int ...Reserve>
10     static void apply(const MeshElements<MeshDimension, IndexType, Real, Reserve...>&mesh,
11                      Functor f) {
12         for (IndexType currElement = 0;
13              currElement < mesh.template getElements<hDim>().size();
14              currElement++){
15             Impl::MeshRun<
16                 hDim,
17                 hDim,
18                 lDim,
19                 MeshDimension,
20                 StartDimension == TargetDimension,
21                 (StartDimension > TargetDimension)
22                 >::run(mesh, currElement, currElement, f);
23         }
24     }
25
26     template<unsigned int MeshDimension, typename Functor,
27             typename IndexType, typename Real, unsigned int ...Reserve>
28     static void apply(IndexType elementIndex,
29                      const MeshElements<MeshDimension, IndexType, Real, Reserve...>&mesh,
30                      Functor f) {
31         static_assert (StartDimension >= TargetDimension,
32             "It is possible to iterate over connected elements"
33             "of a single element for StartDimension > TargetDimension only.");
34
35         Impl::MeshRun<
36             hDim,
37             hDim,
38             lDim,
39             MeshDimension,
40             StartDimension == TargetDimension,
41             (StartDimension > TargetDimension)
42             >::run(mesh, elementIndex, elementIndex, f);
43     }
44 };
45

```

Code listing 2.39 An example of using `MeshConnections`. The `connections` member function of `MeshConnections` returns a `MeshDataContainer` mapping to each element the indexes of the connected elements of the requested dimension. The sequence of indexes does not contain duplicities and the order of the connections can be chosen to be ascending or original.

```

1 // Connected vertices to the cells
2 auto conCellToVert = MeshConnections<3,0>::connections(mesh);
3 for (auto& cell : mesh.getCells()){
4     conCellToVert[cell]; // Vector of connected vertices to the given cell
5 }
6
7 // Connections in the original order in the mesh
8 auto conCellToVertOrig = MeshConnections<3,0,Order::ORDER_ORIGINAL>::connections(mesh);
9 for (auto& cell : mesh.getCells()){
10     conCellToVertOrig[cell]; // Vector of vertices connected to the given cell
11 }
12
13 // Detection of the cells connected to a vertex
14 auto conVertToCell = MeshConnections<0,3>::connections(mesh);
15 for (auto& vert : mesh.getVertices()){
16     conVertToCell[vert]; // Cells connected to the given vertex
17 }

```

is hidden in namespace `Impl`, because it is meant to be used through the class `MeshApply` only. The definition of `MeshApply` is described in Code listing 2.38.

It is important to mention the limitation of using the `MeshApply`. It is not possible to iterate over the connected super-elements directly, i.e., when `TargetDimension` is greater than `StartDimension`. All elements from the dimension `TargetDimension` must be went through to resolve all the connected super-elements. Thus, it is senseless to apply the function to one element only, because there is no advantage over realizing the application for all elements at once. Thence, it is forbidden to use `apply` with `elementIndex` specified if `TargetDimension` is greater than `StartDimension`.

Another important note is that neither `MeshRun` nor `MeshApply` checks whether a function was already called with the current combination of indexes. Therefore, the function might be called more than once for single connected element and the function design must consider this might happen. For example, when `MeshApply<3,0>` is called, one vertex might be reached more than once, because the vertex is connected to more than one face adjacent to a cell. One shall keep in mind this property when using the `MeshApply` functionality.

Finally, we promised to discuss the advantage of the additional template parameter `End`. The addition of the `End` parameter helped us to lower the number of specializations of `MeshRun` needed. If the specialization was based on equality of the `CurrentDimension` and the `TargetDimension` parameters, it would collide with the specialization “`TargetDimension == MeshDimension`“ iterating over the cell boundary. This is because both the specializations have the same priority. The collision would be solved by specifying the priority by adding specialization for the `CurrentDimension` and the `TargetDimension` both equal to the `MeshDimension`. The same would be necessary to be done in the case of edges.

2.4.2 Mesh Connections and the `MeshConnections` Class

The purpose of the function `MeshConnections` is creating the connection matrices $\mathbb{A}_{G_{\mathcal{T}^*}}^{d_1, d_2}$, where the dimensions d_1, d_2 are template parameters named `StartDim` and `TargetDim`. For an example of usage of the `MeshConnections`, see Code listing 2.39. `MeshConnections` provides the functionality of the map “connect” introduced in equation (1.10). Using `MeshApply`, the implementation of `MeshConnections` is very simple.

The only problem to be solved is that the connected elements are visited more than once, because the function `MeshApply` does not care whether a connected element was already visited.

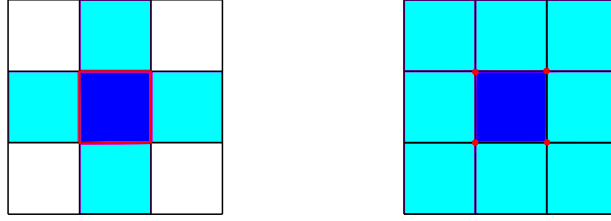


Figure 2.3: Dependence of the neighborhood on the chosen connecting elements. In both figures, the neighborhood of the dark blue square is colored cyan. The left case presents the neighborhood over edges, the right over vertices. The connecting elements are highlighted in red. The effect of the choice of connecting elements is obvious at first sight: the choice of edges resolves in 4-neighborhood, the choice of vertices resolves in 8-neighborhood.

The solution is simple and consists in utilizing the standard template library class `std::set`, which prevents insertion of multiple keys (in our case, the element indexes).

The method starts by allocating the `std::set` on the mesh using `MeshDataContainer<std::set<IndexType>, FromDim>`. In `MeshApply`, the mapped sets are filled with data. Finally, the sets are converted to class `std::vector`, thus the result type is `MeshDataContainer` of vectors. The reason is more compact data structure, because `std::set` is usually implemented as red-black tree [6]. This function returns indexes in ascending order, see Code listing 2.40. The ascending order of indexes is suitable for set operations as `std::set_union` or `set_difference` defined in `algorithm` header.

To provide the possibility to return original the order, which is understood as the order in the `MeshElements` structure, the `MeshConnections` has an additional template parameter which specifies the order of indexes. The default is ascending order. The original order is achieved by using `std::map<IndexType, IndexType>` instead of `std::set<IndexType>`. Alongside each index this map stores in addition the order of the elements as they were added in to the container. Then the data are copied into the vector according to the mapped order index instead of the order of the key set, see Code listing 2.41.

Let us note that there is one more important ordering of indexes. The order is geometrical and it is relevant for the connections from the second dimension to vertices. The indexes of vertices are returned in the order as they appear along the boundary of the surface. The more detailed description is in Section 2.5.

2.4.3 Elements Neighborhood and the MeshNeighborhood Class

Another useful task is to find all neighbors of the given element, e.g., neighboring cells to a cell. This task goes beyond the possibilities of connections, but the concepts of connections and neighborhood are very similar. In terms of graph theory, the neighborhood of a vertex v is a set of graph vertices which are connected with v by an edge. We adapt the definition of neighborhood to respect the mesh geometry. This neighborhood is defined by two dimensions. The first is the connecting dimension d_1 and the second is the connected dimension d_2 . For any element $e \in V_{\mathcal{T}^*}^d$, the set of neighboring elements of dimension d_2 connected by elements of dimension d_1 reads

$$N_{G_{\mathcal{T}^*}}^{d_1, d_2}(e) = \left\{ f \in V_{\mathcal{T}^*}^{d_2} \mid \left(\exists g \in V_{\mathcal{T}^*}^{d_1} \right) \left((g, f) \in E_{\mathcal{T}^*} \wedge (g, e) \in E_{\mathcal{T}^*} \wedge e \neq f \right) \right\}, \quad (2.1)$$

where $d, d_1, d_2 \in \{0, 1, \dots, d_{\mathcal{T}}\}$. Equation (2.1) can be interpreted as: The d_2 -dimensional element $f \in V_{\mathcal{T}^*}^{d_2}$ is a neighbor of $e \in V_{\mathcal{T}^*}^d$ (over dimension d_1) if there exists a path of length 2 from f to e over some d_1 -dimensional element $g \in V_{\mathcal{T}^*}^{d_1}$. The differences of the neighborhood based on the choice of the connecting dimension are shown in Figure 2.3.

Code listing 2.40 The definition of `MeshConnections` for the choice of ascending order of returned indexes of the connected elements. The `connections` member function accepts any mesh as the `MeshElements` class and determines the connections in the mesh according to the template setup of `MeshConnections` class, i.e., the parameters `StartDim` and `TargetDim`. At first, a temporary container for the connections `tmpSet` is prepared to store the connections. The container `tmpSet` is filled using `MeshApply` starting at line 21. Moreover, the contained class `std::set` automatically prevents the duplicities, which means the lambda function considers the possibility of multiple calls with the same combination of parameters `orig` and `connected`. Finally, the `tmpSet` contains the requested connections, that are subsequently copied to a `std::vector`. The ascending order is automatically provided by the `std::set`.

```

1 // From header MesFunctionDefine.h
2 enum Order{
3     ORDER_ASCEND,
4     ORDER_ORIGINAL
5 };
6
7 template<unsigned int StartDim, unsigned int TargetDim, Order order = Order::ORDER_ASCEND>
8 struct MeshConnections {
9
10     // Returns the indexes of the connected elements
11     // from TargetDim to StartDim in ascending order
12     template<unsigned int MeshDimension, typename IndexType,
13             typename Real, unsigned int ...Reserve>
14     static MeshDataContainer<std::vector<IndexType>, StartDim>
15     connections (MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh) {
16
17         // Allocate temporary data
18         MeshDataContainer<std::set<IndexType>, StartDim> tmpSet(mesh);
19
20         // Fill the container
21         MeshApply<StartDim, TargetDim>::apply(
22             mesh,
23             [&tmpSet](IndexType orig, IndexType connected){
24                 tmpSet.template getDataByPos<0>().at(ori).insert(connected);
25             }
26         );
27
28         // Allocate result data
29         MeshDataContainer<std::vector<IndexType>, StartDim> res(mesh);
30
31         // Copy the temporary data into result data
32         for (IndexType i = IndexType(); i < res.template getDataByPos<0>().size(); i++) {
33             res.template getDataByPos<0>()[i].insert(
34                 res.template getDataByPos<0>()[i].begin(),
35                 tmpSet.template getDataByPos<0>()[i].begin(),
36                 tmpSet.template getDataByPos<0>()[i].end());
37         }
38         return res;
39     }
40 };

```

Code listing 2.41 This listing presents the specialization of `MeshConnections` for the parameter `Order` equal to `ORDER_ORIGINAL`. Similarly to the version for ascending order presented in Code listing 2.40, this specialization utilizes a temporary container to automatically resolve duplicities. In contrast to the previous version, it employs `std::map` instead of `std::set`. The reason is to store the information about the order in which the indexes were inserted to the map (see the lambda function starting at line 17). As `tempMap` is filled, the data shall be copied to the result container. Firstly, the result container must be resized to the size of respective map (line 27). Then, in the cycle at line 31 the index is stored at the position given by the order in which it was visited.

```

1  template<unsigned int StartDim, unsigned int TargetDim>
2  struct MeshConnections<StartDim, TargetDim, Order::ORDER_ORIGINAL> {
3      // Returns the indexes of connected elements
4      // from TargetDim to StartDim in original order
5      template<unsigned int MeshDimension, typename IndexType,
6              typename Real, unsigned int ...Reserve>
7      static MeshDataContainer<std::vector<IndexType>, StartDim>
8      connections(
9          MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh
10         ) {
11         // Allocate the temporary mapping
12         MeshDataContainer<std::map<IndexType, IndexType>, StartDim> tempMap(mesh);
13
14         // Fill the container
15         MeshApply<StartDim, TargetDim>::apply(
16             mesh,
17             [&tempMap](IndexType orig, IndexType connected){
18                 IndexType size = tempMap.template getDataByPos<0>().at(orig).size();
19                 tempMap.template getDataByPos<0>().at(orig).insert({connected, size});
20             }
21         );
22
23         // Prepare the result
24         MeshDataContainer<std::vector<IndexType>, StartDim> result(mesh);
25         for (IndexType i = 0; i < mesh.template getElements<StartDim>().size(); i++){
26             // Resize the vector at the position
27             result.template getDataByPos<0>().at(i).resize(
28                 tempMap.template getDataByPos<0>().at(i).size()
29             );
30
31             for(auto& mapElem : tempMap.template getDataByPos<0>().at(i)) {
32                 // Set the index of an element into a vector
33                 // at position corresponding to order it was accessed
34                 result.template getDataByPos<0>().at(i).at(mapElem.second)=mapElem.first;
35             }
36         }
37         return result;
38     }
39 };
40

```


Code listing 2.42 This is an example of working with neighborhood of the mesh elements. The definition of `MeshNeighborhood` is in Code listing 2.43. The first example at line 2 is obtaining vertex to vertex neighborhood, where the connection of vertices is provided by cells, i.e., vertices connected to the same cell are neighboring. The second example presents an algorithm for an extension of neighborhood. The neighborhood is expanded by the neighbors of neighbors. This algorithm consists in joining all the neighborhoods of neighbors into the resulting container. Thanks to the ascending ordering of indexes, it is possible to realize that using `std::set_union`, see the line 14. It is possible to apply the algorithm recursively to determine broader neighborhoods. The last example shows the neighborhood between faces over cells with the original ordering.

```

1 // Getting vertices neighboring to vertices over cells
2 auto nbh = MeshNeighborhood<0,3>::neighbors(mesh);
3
4 // Algorithm for calculating a broader neighborhood
5 MeshDataContainer<std::vector<size_t>,0> nbh2Order(mesh) = nbh;
6 for(auto& vert : mesh.getVertices()){
7     for (auto nvi : nbh2Order[vert]){
8         auto& nVert = mesh.getVertices()[nvi];
9
10        // Prepare temporary container for result of union
11        std::vector<size_t> res;
12
13        // Join the indexes of neighbors of the neighboring vertex
14        std::set_union(nbh2Order[vert].begin(), nbh2Order[vert].end(),
15                      nbh[nVert].begin(), nbh[nVert].end(),
16                      std::inserter(res, res.begin()));
17        // Update the set of indexes
18        nbh2Order[vert] = res;
19    }
20
21    // Erasing the index of vertex from its own neighborhood
22    nbh2Order[vert].erase(lower_bound(nbh2Order[vert].begin(),
23                                    nbh2Order[vert].end(), vert.getIndex()));
24 }
25
26 // Neighborhood between faces over cells ordered as in the mesh
27 auto nbh2 = MeshNeighborhood<2,3,2,ORDER_ORIGINAL>::neighbors(mesh);

```

Code listing 2.43 This code presents the definition of the method calculating the neighborhood. The template parameters `StartDim`, `ConnectingDim` and `ConnectedDim` correspond to the values of d , d_1 , d_2 , respectively, in (2.1). By default, the `ConnectedDim` is equal to `StartDim` and the result order is ascending. The algorithm starts with obtaining connections at lines 15-19. Then, the connections are combined using a loop over connections of connected elements of dimension d_1 , lines 27-39. The correct order is ensured by using `std::set` which additionally prevents duplicities. Note that the restriction that element is not its own neighbor is satisfied by the condition at line 33. Finally, at line 41, the result is copied from the temporary set into the resulting vector. The complete description is in Section 2.4.3.

```

1  template <unsigned int StartDim,
2      unsigned int ConnectingDim,
3      unsigned int ConnectedDim = StartDim,
4      Order order = Order::ORDER_ASCEND>
5  class MeshNeighborhood{
6  public:
7      template<unsigned int MeshDimension,
8          typename IndexType,
9          typename Real,
10         unsigned int ...Reserve>
11     static MeshDataContainer<std::vector<IndexType>, StartDim> neighbors(
12         MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh
13     ) {
14         // Prepare the connections
15         MeshDataContainer<std::vector<IndexType>, StartDim> result(mesh);
16         auto firstConnections =
17             MeshConnections<StartDim, ConnectingDim, order>::connections(mesh);
18         auto secondConnections =
19             MeshConnections<ConnectingDim, ConnectedDim, order>::connections(mesh);
20         // Join the connections
21         for (IndexType elementIndex = 0;
22             elementIndex < mesh.template getElements<StartDim>().size();
23             elementIndex++) {
24
25             std::set<IndexType> tmpResultSet;
26
27             for (IndexType& firstConnectedElem :
28                 firstConnections.template getDataByPos<0>().at(elementIndex)){
29
30                 for (IndexType& neighborIndex :
31                     secondConnections.template getDataByPos<0>().at(firstConnectedElem)){
32                     // Do not set the element as its own neighbor
33                     if (StartDim == ConnectedDim && elementIndex == neighborIndex) {
34                         continue;
35                     } else {
36                         tmpResultSet.insert(neighborIndex);
37                     }
38                 }
39             }
40             // Copy the result into result container
41             result.template getDataByPos<0>().at(elementIndex).insert(
42                 result.template getDataByPos<0>().at(elementIndex).begin(),
43                 tmpResultSet.begin(),
44                 tmpResultSet.end()
45             );
46         }
47
48         return result;
49     }
50 };
51

```

Code listing 2.44 The specialization of `MeshNeighborhood` class for `Order = ORDER_ORIGINAL`. Similarly to `MeshConnection` with this choice of ordering, this implementation utilizes `std::map` to remember the order of added elements indexes. Other thoughts are the same as in the default case described in Code listing 2.43.

```

1  template <unsigned int StartDim,
2          unsigned int ConnectingDim,
3          unsigned int ConnectedDim>
4  class MeshNeighborhood<StartDim,
5                        ConnectingDim,
6                        ConnectedDim,
7                        Order::ORDER_ORIGINAL> {
8  public:
9      template<unsigned int MeshDimension,
10             typename IndexType,
11             typename Real,
12             unsigned int ...Reserve>
13      static MeshDataContainer<std::vector<IndexType>, StartDim> neighbors(
14          MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh
15      ) {
16          // Prepare the connections
17          MeshDataContainer<std::vector<IndexType>, StartDim> result(mesh);
18          auto firstConnections =
19              MeshConnections<StartDim, ConnectingDim, ORDER_ORIGINAL>::connections(mesh);
20          auto secondConnections =
21              MeshConnections<ConnectingDim, ConnectedDim, ORDER_ORIGINAL>::connections(mesh);
22          // Join the connections
23          for (IndexType elementIndex = 0;
24              elementIndex < mesh.template getElements<StartDim>().size();
25              elementIndex++) {
26
27              std::map<IndexType, IndexType> tempResultMap;
28
29              for (IndexType& firstConectedElem :
30                  firstConnections.template getDataByPos<0>().at(elementIndex)){
31
32                  for (IndexType& neighborIndex :
33                      secondConnections.template getDataByPos<0>().at(firstConectedElem)){
34                      // Do not set the element as its own neighbor
35                      if (StartDim == ConnectedDim && elementIndex == neighborIndex) {
36                          continue;
37                      } else {
38                          IndexType pos = tempResultMap.size();
39                          tempResultMap.insert({neighborIndex, pos});
40                      }
41                  }
42              }
43          // Copy the result into result container
44          result.template getDataByPos<0>().at(elementIndex).resize(
45              tempResultMap.size()
46          );
47
48          for(std::pair<const IndexType, IndexType>& mapElem : tempResultMap) {
49              result.template getDataByPos<0>().at(elementIndex).at(mapElem.second) =
50                  mapElem.first;
51          }
52      }
53
54      return result;
55  }
56
57 };

```

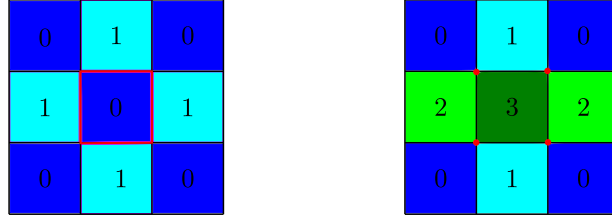


Figure 2.4: An example of coloring of the mesh cells. In both cases a greedy algorithm was used. The difference consists in the choice of the connecting dimension, which influences the shape of the neighborhood of cells. For an example, see Figure 2.3.

The realization in GTMesh is utilizing three dimension parameters. The first is the dimension of elements to find the neighborhood for. The neighborhood will be found for all elements of the dimension named `StartDim`. The second is the connecting dimension `ConnectingDim`, i.e. d_1 in (2.1). Finally, the third is the connected dimension `ConnectedDim`, i.e. d_2 in (2.1). As in `MeshConnections`, the returned container is a `MeshDataContainer` mapping a vector of indexes of the neighbors to each mesh element of `StartDim` dimension. This functionality also provides the choice of ordering the indexes of neighbors. The default value of ordering is ascending order, see the definition of the `MeshNeighborhood` in Code listing 2.43. The `MeshNeighborhood` class has a public static member function `neighbors`, which realizes the algorithm. The detection of neighbors utilizes two connections: one from `StartDim` to `ConnectingDim` and the other from `ConnectingDim` to `ConnectedDim`. By combination of those two connections, the neighborhood of elements of `StartDim` dimension to `ConnectedDim` over `ConnectingDim` is realized. Additionally, when the `StartDim` is equal to `ConnectedDim`, the index of the origin element must not be in the neighborhood, therefore it is omitted using the condition at line 21 in Code listing 2.43. This function returns the indexes of the neighboring elements sorted, which can be used with advantage to join the neighborhoods of neighboring elements. This way, more distant neighbors can be obtained. An example is presented in Code listing 2.42.

The specialization for the original order is achieved as in the case of `MeshConnections`. The function utilizes a map to remember the order of visited indexes, as shown in Code listing 2.44.

2.4.4 Mesh Coloring and the ColorMesh Class

The last of the algorithms related to the graph representation of the mesh topology is the coloring algorithm. The problem consists in proper coloring of the elements of dimension d according to connections of dimension d_1 [31]. In terms of graph theory, it is a proper vertex coloring of the graph $G = (V, E)$ where

$$\begin{aligned}
 V &= V_{\mathcal{J}^*}^d, \\
 E &= \left\{ (e, v) \in \left(V_{\mathcal{J}^*}^d \right)^2 \mid v \in N_{G_{\mathcal{J}^*}}^{d_1, d}(e) \right\}.
 \end{aligned}
 \tag{2.2}$$

In other words, the vertices of the graph G are mesh elements of dimension d and two vertices are connected by edge, if the corresponding elements are both connected to the same element of dimension d_1 .

Definition 6. Let $G = (V, E)$ be a graph. Vertex coloring is a mapping $\phi : V \mapsto \mathbb{N}$ and the values of ϕ are called colors. The coloring ϕ is called proper, iff

$$(\forall e \in V) (\forall v \in N_G(e)) (\phi(e) \neq \phi(v)),
 \tag{2.3}$$

where $N_G(e) = \{f \in V \mid (e, f) \in E\}$ is the set of neighbors of the vertex $e \in V$ in G .

The problem can be demonstrated on the example of coloring the cells with respect to vertices, i.e. $d = d_{\mathcal{T}}$ and $d_1 = 0$. This means, the cells are colored such that all cells sharing the same vertex have a unique color, see Figure 2.4.

The simplest algorithm to find a proper coloring of a graph is a greedy algorithm. The algorithm is the following:

1. For each vertex $e \in V$:
 - (a) loop over the neighboring vertices $v \in N_G(e)$ of e and keep track of the used colors,
 - (b) select the lowest color not present among the neighbors and assign it to e .

The greedy algorithm might generate uneven distribution of colors with some colors occurring only sparsely. If a more even distribution is desired, a random re-selection of colors can be applied. This update of the given colors may cause more even distribution of the colors. The algorithm of the random update is very similar to the greedy one and utilizes previously calculated coloring. Hence, this algorithm has an information about the number of colors needed for proper coloring and it is able to select another color where possible. The algorithm reads:

1. For each vertex $v \in V$:
 - (a) loop over the neighboring vertices $v \in N_G(e)$ of e and keep track of the used colors,
 - (b) select randomly a new color of the vertex v or keep the old one.

Now let us move to the description of the implementation. GTMesh provides the coloring algorithm through the class `MeshColoring` with a public static member function `color` which accepts a reference to an instance of `MeshElements`. The method then calls the `color` member function of the class `_MeshColoring`, which realizes the algorithm according to the template setup. The template parameters are d as `ColoredDim` and d_1 as `ConnectingDim` and `Method` to select the greedy algorithm or the one with the random update.

The utilized algorithm advantageously uses the structure of the mesh to simulate the $G_{\mathcal{T}^*}$. The advantage consists in utilizing the elements of dimension d_1 as a storage of the colors of the neighboring elements of dimension d . This way, the occupied colors by the neighbors of $v \in \mathcal{T}^d$ can be determined by looping over the $N^{d_1}(v)$. The chosen data structure to store the occupied colors among the neighbors ($N^d(e)$) of $e \in \mathcal{T}^{d_1}$ is `std::valarray<bool>` where the position corresponds to the index of the color and value determines, whether the color is already used among the neighbors. The advantage of this container is the possibility to detect the free colors for elements by simple boolean operations, see the line 31 in Code listing 2.45. This approach requires an update of the occupied colors after color is set to v .

The implemented greedy version algorithm is the following:

1. Prepare the containers (`MeshDataContainer`) mapping colors to $v \in \mathcal{T}^d$ and `std::valarray<bool>` to $e \in \mathcal{T}^{d_1}$,
2. for each element v :
 - (a) determine the occupied colors among $N^{d,d_1}(v)$ by looping over $N^{d_1}(v)$,
 - (b) select the free color with the lowest index (resize the `std::valarray<bool>` mapped to \mathcal{T}^{d_1} if all colors are occupied),
 - (c) set true value at position corresponding to the selected color at all $N^{d_1}(v)$.

Because the number of colors needed is not known in ahead, it is possible to the number of colors exceeds the reserved number of colors. In such case the reserve for the number of colors is doubled in order to be able to handle more colors (see lines 40-50 in Code listing 2.45).

Code listing 2.45 The implementation of the greedy mesh coloring algorithm described in Section 2.4.4. The method firstly prepares the container for the result and `colorReserve` number of colors. Then the container of attached colors to connecting elements is together with connections from `ColoredDim` to `ConnectingDim` are prepared. The container to hold the attached colors is `std::valarray<bool>`, e.g., `true` on position 2 means the color number 2 is already used. The convenient use of `std::valarray<bool>` for aggregation of colors is at lines 29-32. When the list of free colors is obtained, the first free color is chosen then. If there is not free color in the list, then the number of considered colors is doubled. Finally, the chosen color is set as the color of the current element and reserved in the `attachedColors` container.

```

1  template<unsigned int ColoredDim,
2      unsigned int ConnectingDim,
3      ColoringMethod Method = METHOD_GREEDY,
4      bool Descend = (ColoredDim > ConnectingDim)>
5  struct _MeshColoring {
6      template<unsigned int MeshDimension, typename IndexType, typename Real, unsigned int ...Reserve>
7      static MeshDataContainer<unsigned int, ColoredDim> color(
8          MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh
9      ) {
10
11         // resulting container of colors
12         MeshDataContainer<unsigned int, ColoredDim> result(mesh);
13         // Setup the initial reserved number of colors
14         unsigned int colorReserve = 16;
15
16         // Allocate the containers for attached colors to "edges"
17         MeshDataContainer<std::valarray<bool>, ConnectingDim> attachedColors;
18         attachedColors.allocateData(mesh, std::valarray<bool>(false, colorReserve));
19
20         // Create the connections from ColoredDim to ConnectingDim
21         auto connections = MeshConnections<ColoredDim, ConnectingDim>::connections(mesh);
22
23         // Loop over all mesh elements in ColoredDim
24         for (IndexType elementIndex = 0;
25             elementIndex < mesh.template getElements<ColoredDim>().size();
26             elementIndex++){
27
28             // Gather the free colors over adjacent elements from ConnectingDim
29             std::valarray<bool> freeColors(true, colorReserve);
30             for (IndexType element : connections.template getDataByPos<0>().at(elementIndex)){
31                 freeColors &= !attachedColors.template getDataByPos<0>().at(element);
32             }
33
34             // Select the first possible color
35             unsigned int selectedColor = 0;
36             while (!freeColors[selectedColor]) {
37                 selectedColor++;
38                 // If the number of colors exceeds the number of
39                 // allocated bits, then allocate memory for twice as many colors
40                 if (selectedColor == freeColors.size()){
41                     colorReserve *= 2;
42                     for (std::valarray<bool>& attColor : attachedColors.template getDataByPos<0>()){
43                         std::valarray<bool> newAttColor(false, colorReserve);
44                         for (size_t i = 0; i < attColor.size(); i++){
45                             newAttColor[i] = attColor[i];
46                         }
47                         attColor.swap(newAttColor);
48                     }
49                     break;
50                 }
51             }
52             // Set the selected color to the element
53             result.template getDataByPos<0>().at(elementIndex) = selectedColor;
54
55             // Set the selected color as not free at adjacent edges
56             for (IndexType element : connections.template getDataByPos<0>().at(elementIndex)){
57                 DBGTRY(attachedColors.template getDataByPos<0>().at(element)[selectedColor] = true;);
58             }
59         }
60         return result;
61     }
62 };

```

Code listing 2.46 The specialization of the `_MeshColoring` for `Descend == true`, i.e., `ColoredDim` higher than `ConnectingDim`. The iterations over sub-elements can be resolved using the `MeshElements` data structure without prior declaration of the connections. Therefore, `MeshApply` can be utilized. Other parts of the code are the same as in the generic case in Code listing 2.45.

```

1  template<unsigned int ColoredDim, unsigned int ConnectingDim>
2  struct _MeshColoring <ColoredDim, ConnectingDim, METHOD_GREEDY, true> {
3
4      // The implementation for choice of Descend == true
5      template<unsigned int MeshDimension, typename IndexType,
6              typename Real, unsigned int ...Reserve>
7      static MeshDataContainer<unsigned int, ColoredDim> color(
8          MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh
9      ) {
10         // Prepare the containers for the algorithm
11         MeshDataContainer<unsigned int, ColoredDim> result(mesh);
12         unsigned int colorReserve = 16;
13         MeshDataContainer<std::valarray<bool>, ConnectingDim>
14             attachedColors(mesh, std::valarray<bool>(false, colorReserve));
15
16         for (IndexType elementIndex = 0;
17             elementIndex < mesh.template getElements<ColoredDim>().size();
18             elementIndex++){
19
20             std::valarray<bool> freeColors(true, colorReserve);
21             MeshApply<ColoredDim, ConnectingDim>::apply(
22                 elementIndex,
23                 mesh,
24                 [&freeColors, &attachedColors](IndexType, IndexType element){
25                     freeColors &= !attachedColors.template getDataByPos<0>().at(element);
26                 }
27             );
28
29             // Select the first possible colour
30             // the same as lines 34-51 in Code listing 2.45
31
32
33             result.template getDataByPos<0>().at(elementIndex) = selectedColor;
34             MeshApply<ColoredDim, ConnectingDim>::apply(
35                 elementIndex,
36                 mesh,
37                 [selectedColor, &attachedColors](IndexType, IndexType element){
38                     attachedColors.template getDataByPos<0>()[element][selectedColor] =
39                         true;
40                 });
41         }
42     }
43 };

```

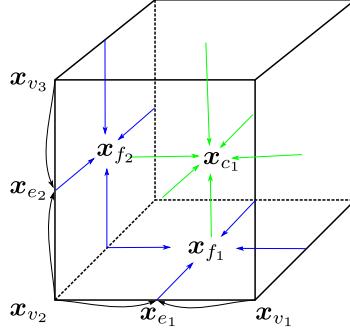


Figure 2.5: An example of the chosen algorithm for computing the element centers in a simple 3D mesh. The center point of an element is calculated as an average of the center points of its sub-elements. For example, the cell center point denoted \mathbf{x}_c is the average of the centers of all connected faces, denoted \mathbf{x}_{f_i} .

Moreover, it is possible to employ an optimization for `ConnectingDim` lower than `ColoredDim`. The connecting elements can be iterated directly and there is no need to explicitly evaluate the connections. Code listing 2.45 presents the definition of `_MeshColoring`, which demonstrates the greedy method and `ConnectingDim` higher than `ColoredDim`. The presented algorithm is generic and works for any choice of the dimensions.

In the case of `ConnectingDim` is lower than `ColoredDim`, the for cycles over connecting elements (lines 30 and 56 in Code listing 2.45) are replaced by `MeshApply`, see the specialization in Code listing 2.46.

The class `MeshColoring` with method `color` wraps the implementation hidden in `_MeshColoring` in namespace `Impl`.

2.4.5 Element Center Calculation

The previous algorithms were working with the mesh as a topological object, i.e., they did not work with the coordinates of vertices. Now we will describe the algorithms that calculate some significant geometrical properties of the mesh. We begin with the algorithm calculating center points of all objects in the mesh.

The center of each element is calculated as an average of the positions of centers of the connected sub-elements, e.g., the center of a cell is an average of centers of the cell's faces. The advantage of this approach consists in reduction of the depth of iteration over the mesh, see the example of the algorithm in Figure 2.5. Because the algorithm uses the previously calculated values, the computation might speed up against the calculation of the center as the average of the positions of the connected vertices. For example, in the case of dimension 2 it is sufficient to visit the sub-elements of dimension 1. The scheme of the algorithm is the following:

1. set the dimension $d = 1$,
2. for all elements $e \in \mathcal{T}^d$ calculate the center of e as $\mathbf{x}_e = \frac{1}{|N^{d-1}(e)|} \sum_{f \in N^{d-1}(e)} \mathbf{x}_f$ by a loop over sub-elements of e ,
3. if $d < d_{\mathcal{T}}$ then $d = d + 1$ and go to step 2, else stop the algorithm and return the result.

In a 2D mesh this algorithm returns the same result as the average of the positions of the connected vertices. However, in a 3D mesh the cell centers calculated by this algorithm and average of positions of the connected vertices may differ when the number of vertices of faces differs.

In GTMesh the calculation of the element centers is provided by the `computeCenters` function template. Because the function `computeCenters` has to the loop over the mesh dimensions,

which is a template parameter, it must utilize a template class `_ComputeCenters`, which realizes the loop using template recursion. This concept was already presented in `MeshApply` in Section 2.4.1. In contrast to `MeshRun`, the iteration over mesh dimensions is upward. Moreover, this is the first time we face the problem of returning results mapped to more dimensions. The dimensions to map the result to even depends on the dimension of the mesh. For this purpose, the concept `MakeMeshDataContainer` was developed (Section 2.3.3). For example, the return type in the case of a 2D mesh is `MeshDataContainer<Vertex<2, Real>, 1, 2>`, where the `Vertex` class is presented in Code listing 2.9.

Let us begin with the description of the class `_ComputeCenters` and its specializations. The class `_ComputeCenters` has one public static function `compute`. This function has two parameters, the first is the container to store the result to and the second is the mesh to be processed. The class `_ComputeCenters` has three template parameters which can be used in the `compute` member function. The first parameter is the currently processed dimension named `CurrentDimension`. The second parameter is `MeshDimension` which specifies the number of coordinates of the vertices and also stops the recursion when `CurrentDimension` reaches `MeshDimension`. The last parameter specifies the algorithm to be applied to the computation. In this section, we will describe only the default version. The other one will be presented in Section 2.7.

Let us begin from the specialization which is to be applied first. It is the specialization for the parameter `CurrentDimension` equal to 1. In other words, it is the computation of the centers of edges. The implementation is shown in Code listing 2.48. The `compute` member function firstly calculates the center points as an average of the two vertices defining the edge. Notice the formula for calculation of the average at lines 23 and 24 in Code listing 2.48. Despite using `MeshElement` with dimension 0, it is possible to treat it as the `Vertex` class. After preparing the centers of edges, the function calls `compute` of `_ComputeCenters` with `CurrentDimension` set to 2. The centers container is passed to the called function, which can use the prepared centers for further calculation.

Next, we describe the generic definition `_ComputeCenters`, which is applied when `CurrentDimension` is greater than 1 and less than `MeshDimension`. The implementation of the `compute` method utilizes a generic algorithm to sum the centers of the sub-elements and divide it by the number of sub-elements. The iteration over sub-elements can be comfortably realized using `MeshApply`. For details, see Code listing 2.47. After the centers are prepared, the function again calls `compute` of `_ComputeCenters` with an increased dimension.

Finally, when `CurrentDimension` reaches `MeshDimension`, the last specialization is to be used. This specialization calculates the centers of cells by the same algorithm utilized in the generic case. In other words, the centers of the cells are calculated from the previously calculated centers of faces. Thanks to the generality of the `MeshApply` concept, the implementation of the averaging algorithm might be the same as in the generic case. In contrast to the generic case, in this specialization the recursion stops, i.e., there is no call of `_ComputeCenters` after the computation. Then the algorithm stops and the result is returned to the caller by the parameter `centers`.

As in the case of the previous algorithms, the GTMesh library provides a function wrapper, which utilizes `_ComputeCenters` hidden in the namespace `Impl`. The function is named `computeCenters` and accepts one parameter, which is the mesh to work with, and one template parameter, the computation method. The rest of template parameters are deduced. Then the function prepares the container corresponding to the mesh dimension and calls the `compute` member function of `_ComputeCenters<1, MeshDimension>`. Finally, the calculated result is returned. See the implementation in Code listing 2.50.

Code listing 2.47 The definition of the `_ComputeCenters` class. In the context of the following specializations, this definition is used for elements with generic connection to sub-elements, e.g., face in 3D mesh. The member function named `compute` returns the result in the parameter `centers` passed by reference. The implementation assumes that the container is already allocated and the centers mapped to lower dimensions than `CurrentDimension` are already prepared. Then, the function calculates the centers in the currently processed dimension. Finally, the calculation is passed to the next dimension. Note that `MakeMeshDataContainer_t` together with `make_custom_integer_sequence_t` serves as automatic generator of the return type, as discussed in Section 2.3.3. The template parameter `Method` changes the computation for meshes with non-planar faces. This will be further discussed in Section 2.7.

```

1  template <unsigned int CurrentDimension, unsigned int MeshDimension,
2          ComputationMethod Method = ComputationMethod::DEFAULT>
3  struct _ComputeCenters {
4      template <typename IndexType, typename Real, unsigned int ...Reserve>
5      static void
6      compute(
7          MakeMeshDataContainer_t<
8              Vertex<MeshDimension, Real>,
9              make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>
10         >& centers, // [out]
11         const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh) {
12
13         // Container of centers to be calculated
14         auto& elemCenters = centers.template getDataByDim<CurrentDimension>();
15
16         // Container of centers used for calculation
17         auto& subElemCenters = centers.template getDataByDim<CurrentDimension - 1>();
18
19         for (IndexType i = 0;
20             i < mesh.template getElements<CurrentDimension>().size();
21             i++){
22             // Calculate an average for a single element of dimension CurrentDimension
23             Real subElemCnt = 0;
24             MeshApply<CurrentDimension, CurrentDimension - 1>::apply(
25                 i,
26                 mesh,
27                 [&elemCenters, &subElemCenters, &subElemCnt]
28                 (IndexType elementIndex, IndexType subelementIndex){
29                     elemCenters.at(elementIndex) += subElemCenters.at(subelementIndex);
30                     subElemCnt++;
31                 }
32             );
33
34             elemCenters.at(i) /= subElemCnt;
35         }
36         // Continue with the computation for higher dimension
37         _ComputeCenters<CurrentDimension + 1, MeshDimension, Method>::compute(centers, mesh);
38     }
39 };

```

Code listing 2.48 The specialization of `_ComputeCenters` for the case of edges, i.e., `CurrentDimension` equal to 1. The centers are calculated as an average of the two vertices of the edge. Then, the calculation continues for the second dimension.

```

1  template <unsigned int MeshDimension, ComputationMethod Method>
2  struct _ComputeCenters<1, MeshDimension, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4      static void
5      compute(
6          MakeMeshDataContainer_t<
7              Vertex<MeshDimension, Real>,
8              make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>>& centers,
9          const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh) {
10
11         // Get the reference to computed centers std::vector<Vertex<MeshDimension, Real>>
12         auto& edgeCenters = centers.template getDataByDim<1>();
13
14         // Calculate centers for all edges in the mesh
15         for (IndexType edgeIndex = 0;
16             edgeIndex < mesh.template getElements<1>().size();
17             edgeIndex++) {
18
19             const auto& edge = mesh.getEdges().at(edgeIndex);
20
21             // Utilization of MeshElement<0,...> as Vertex
22             edgeCenters.at(edgeIndex) =
23                 (mesh.template getElements<0>().at(edge.getVertexAIndex()) +
24                  mesh.template getElements<0>().at(edge.getVertexBIndex())) * 0.5;
25         }
26
27         _ComputeCenters<2, MeshDimension, Method>::compute(centers, mesh);
28     }
29 };

```

Code listing 2.49 The final specialization of `_ComputeCenters` for cells, i.e., `CurrentDimension == MeshDimension`. This specialization calculates the centers of cells using previously prepared centers of faces and terminates the recursion. After the function ends, the control is gradually returned to the calling function that has the result stored in the container it passed as `centers`.

```

1  template < unsigned int MeshDimension,
2          ComputationMethod Method >
3  struct _ComputeCenters<MeshDimension, MeshDimension, Method>{
4      template <typename IndexType, typename Real, unsigned int ...Reserve>
5          static void
6          compute(
7              MakeMeshDataContainer_t<
8                  Vertex<MeshDimension, Real>,
9                  make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>>& centers,
10             const MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh) {
11
12             // Container of centers to be calculated
13             auto& elemCenters = centers.template getDataByDim<MeshDimension>();
14
15             // Container of centers used for calculation
16             auto& subElemCenters = centers.template getDataByDim<MeshDimension - 1>();
17
18             for (IndexType i = 0; i < mesh.template getElements<MeshDimension>().size(); i++){
19                 // Calculate the average for a single cell
20                 Real subElemCnt = 0;
21                 MeshApply<MeshDimension, MeshDimension - 1>::apply(
22                     i,
23                     mesh,
24                     [&elemCenters, &subElemCenters, &subElemCnt]
25                     (IndexType elementIndex, IndexType subelementIndex){
26                         elemCenters.at(elementIndex) += subElemCenters.at(subelementIndex);
27                         subElemCnt++;
28                     }
29                 );
30
31                 elemCenters.at(i) /= subElemCnt;
32             }
33         }
34         // Here the computation is terminated
35         // and the result is stored in the "centers" container
36     };

```

Code listing 2.50 The definition of the global function `computeCenters`. This function wraps the `_ComputeCenters` functionality and ensures the correct call of `_ComputeCenters` according to the given mesh parameter. The `_ComputeCenters` class is not designed to be called directly by the user, therefore it is defined in the `Impl` namespace. Finally, the resulting `MeshDataContainer` with the centers of all mesh elements is returned.

```

1  template < ComputationMethod Method,
2          unsigned int Dimension,
3          typename IndexType,
4          typename Real,
5          unsigned int ...Reserve >
6  MakeMeshDataContainer_t< Vertex < Dimension, Real >,
7                          make_custom_integer_sequence_t< unsigned int, 1, Dimension > >
8  computeCenters( const MeshElements< Dimension, IndexType, Real, Reserve... >& mesh ){
9
10     MakeMeshDataContainer_t<
11         Vertex<Dimension, Real>,
12         make_custom_integer_sequence_t< unsigned int, 1, Dimension >
13     > centers(mesh);
14
15     Impl::_ComputeCenters< 1, Dimension, Method >::compute( centers, mesh );
16
17     return centers;
18 }

```

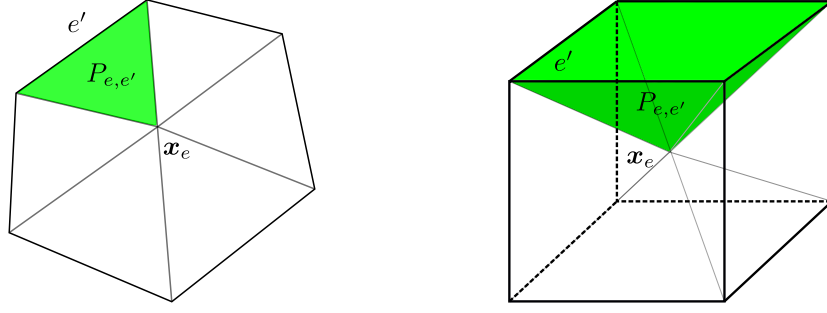


Figure 2.6: An example of computation of measures of polygonal or polyhedral elements.

2.4.6 Element Measure Calculation and the computeMeasures Function

In this section, the calculation the measures of mesh elements is described.

The problem is to develop an algorithm which calculates the measures of all elements in the mesh with dimension grater than 0. Since the elements of the mesh are generic polytopes, the algorithm must be able to handle such generic objects. We assume that every polytope is a star domain with respect to the center of the polytope [32]. Such polytope can be subdivided into pyramids with bases formed by their sub-elements and a common top formed by the center point of the element. We denote such pyramid $P_{e,e'}$, where $e \in \mathcal{T}^d$ and $e' \in \mathcal{T}^{d-1}$, e' is a sub-element of e and $1 < d \leq d_{\mathcal{T}}$. Let us recall that $d_{\mathcal{T}}$ is the dimension of the mesh. An example of subdivision of an element into pyramids is presented in Figure 2.6. Therefore, for any element $e \in \mathcal{T}^d$

$$e = \bigcup_{e' \in N^{d-1}(e)} P_{e,e'} \quad (2.4)$$

holds, where $N^{d-1}(e)$ are the elements of dimension $d - 1$ connected to the element e , see Definition 4. The measure of an element e is given by the sum of the measures of the pyramids, i.e., the formula for the measure of the element $e \in \mathcal{T}^d$, $1 < d \leq d_{\mathcal{T}}$ reads:

$$m(e) = \sum_{e' \in N^{d-1}(e)} m(P_{e,e'}). \quad (2.5)$$

For each mesh vertex $v \in \mathcal{T}^0$, let us denote its geometrical position in space by \mathbf{v} . Then, the measure of the pyramid $P_{e,e'}$ reads:

$$m(P_{e,e'}) = \frac{1}{d} m(e') \text{dist}(V_{e'}, \mathbf{x}_e) = \frac{1}{d} m(e') \text{dist}(S_{e'}, \mathbf{x}_e - \mathbf{v}_1), \quad (2.6)$$

where $V_{e'}$ is a linear manifold of dimension $d - 1$ containing the element e' and

$$S_{e'} = V_{e'} - \mathbf{v}_1,$$

where $v_1 \in N^0(e')$ is a vertex element connected to e' . Note that the $m(e')$ is the measure of the pyramid base and $\text{dist}(V_{e'}, \mathbf{x}_e)$ is the height of the pyramid $P_{e,e'}$ in equation (2.6). Lastly, in the case of edges, i.e. $e \in \mathcal{T}^1$, the length is given by:

$$m(e) = |\mathbf{v}_A - \mathbf{v}_B|, \quad (2.7)$$

where $\{v_A, v_B\} = N^0(e)$.

From the previous analysis, we have obtained the formulae (2.5) and (2.6), which require the measures of the sub-elements to calculate the measure of the current element. However, thanks to equation (2.7), the recursion stops once we know the measure (length) of edges. Therefore, it is sufficient to calculate the height of the pyramid, i.e., the distance of \mathbf{x}_e from the linear

Code listing 2.51 The definition of the `_ComputeMeasures` class. Since GTMesh provides the calculation for 2D and 3D meshes only, the general definition causes an error in the case of application of `_ComputeMeasures` on a mesh with dimension higher than 3. The individual cases will be handled by the respective specializations.

```

1  template <
2      unsigned int CurrentDimension,
3      unsigned int MeshDimension,
4      ComputationMethod Method = DEFAULT
5  >
6  struct _ComputeMeasures{
7      template <typename IndexType, typename Real, unsigned int ...Reserve>
8      static void compute(
9          MakeMeshDataContainer_t<
10             Real,
11             make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>
12         >&,
13         MeshElements<MeshDimension, IndexType, Real, Reserve...>&){
14         // If the MeshDimension is
15         static_assert (
16             MeshDimension <= 3,
17             "The measure computation for dimension higher than 3 is not implemented yet."
18         );
19     }
20 };

```

manifold defined by the vertices of the e' . Furthermore, the origin can be set to \mathbf{v}_1 , the first vertex connected to e' . Then, $S_{e'}$ has the form:

$$S_{e'} = \text{span}(\mathbf{v}_2 - \mathbf{v}_1, \dots, \mathbf{v}_{n_{e'}} - \mathbf{v}_1), \quad (2.8)$$

where $\{\mathbf{v}_1, \dots, \mathbf{v}_{n_{e'}}\} = N^0(e')$. Finally, the problem is to determine

$$\text{dist}(S_{e'}, \mathbf{x}_e - \mathbf{v}_1). \quad (2.9)$$

Let us recall that this algorithm is limited to polytopes with star domain only.

So far, the GTMesh library offers the method for calculation of measures in 2D and 3D unstructured meshes. The measure calculation is realized by the class `_ComputeMeasures` with the `compute` member function. Similarly to the previous functions, the implementation is not designed to be used by the user directly, hence it is hidden in the `Impl namespace`.

The computation of edge length is the same in both 2D and 3D. The general definition checks whether the dimension of the given mesh is lower than 3, otherwise it causes an error with description that such dimension is not allowed (see Code listing 2.51).

The specialization of the calculation of the length of the edges utilizes the fact that the class `Vertex` behaves as a vector. That means, the subtraction of two vertices is a new vertex (vector). Finally, it remains to calculate the Euclidean norm by calling the `normEuclid` member function on the difference of vertices A and B . The implementation is in Code listing 2.52.

Next, when the measures of edges are computed, the computation is passed to the function computing the measures of elements with dimension 2. The computation differs according to the mesh dimension, therefore the cases will be presented separately. Let us begin with the easier one, which is the 2D case.

Calculation of Cell Measures in a 2D Mesh In a 2D mesh, the measure of the elements can be simplified. The point is that the pyramids are in fact triangles. Therefore, we can utilize the commonly known relation calculating the measure of a triangle from coordinates of its vertices. For the triangle constructed from center point of the cell c (\mathbf{x}_c) and vertices of edge e , i.e. $\mathbf{v}_A, \mathbf{v}_B$, the formula reads

$$m(P_{c,e}) = \frac{1}{2} ((x_{c,1} - v_{A,1})(v_{B,2} - v_{A,2}) - (x_{c,2} - v_{A,2})(v_{B,1} - v_{A,1})). \quad (2.10)$$

Code listing 2.52 The specialization of `_ComputeMeasures` calculating the length of the edges. The length is calculated according to equation (2.7). The algorithm applies the `normEuclid` member function to the difference of the vertices. Note that this is possible because subtraction of two `Vertex` instances is a `Vertex`. Then the computation is passed to the next dimension.

```

1  template <unsigned int MeshDimension, ComputationMethod Method>
2  struct _ComputeMeasures<1, MeshDimension, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4          static void compute(
5              MakeMeshDataContainer_t<
6                  Real,
7                  make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>
8              >& measures,
9              MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh){
10
11          auto& edgeLengths = measures.template getDataByDim<1>();
12
13          for (IndexType edgeIndex = 0;
14              edgeIndex < mesh.template getElements<1>().size();
15              edgeIndex++) {
16
17              const auto& edge = mesh.getEdges().at(edgeIndex);
18
19              const auto& vertA = mesh.getVertices().at(edge.getVertexAIndex());
20              const auto& vertB = mesh.getVertices().at(edge.getVertexBIndex());
21              // Calculate the distance between the vertices
22              edgeLengths.at(edgeIndex) = (vertA - vertB).normEuclid();
23          }
24
25          _ComputeMeasures<2, MeshDimension>::compute(measures, mesh);
26      }
27 };

```

The implementation for this setup, i.e., when both `MeshDimension` and `CurrentDimension` are equal to 2, is presented in Code listing 2.53. The algorithm calculates the measure of a cell as the sum of the areas of the triangles, where the area of a triangle in 2D is given by the formula (2.10). The advantage of this approach is the speed of the computation, because the formula is very simple and does not require any calculation of distance.

Measures Calculation in a 3D Mesh Unlike the 2D case, the 3D one is more complicated. The reason is that there is no such formula as (2.10), hence it is necessary to calculate the distance (2.9). The distance of a point \mathbf{x}_f from a line $V_e = \mathbf{v}_A\mathbf{v}_B$ in 3D. We obtain the point \mathbf{w} on the line $\mathbf{v}_A\mathbf{v}_B$ such that the vector $\mathbf{w} - \mathbf{x}_f$ is perpendicular to the line as

$$\mathbf{w} = \mathbf{v}_A - \frac{(\mathbf{v}_A - \mathbf{x}_f) \cdot (\mathbf{v}_B - \mathbf{v}_A)}{(\mathbf{v}_B - \mathbf{v}_A) \cdot (\mathbf{v}_B - \mathbf{v}_A)} (\mathbf{v}_B - \mathbf{v}_A). \quad (2.11)$$

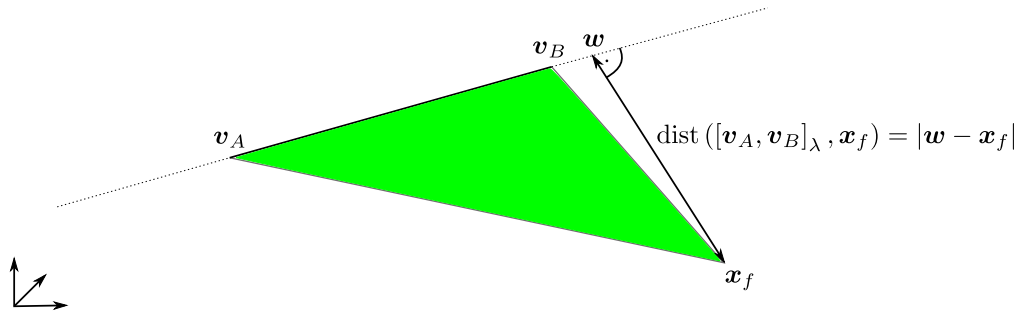


Figure 2.7: Calculation of the height of a triangle given by one edge and a face center in 3D space.

Code listing 2.53 The implementation of the specialization of the `_ComputeMeasures` class calculating the measures of cells in a 2D mesh.

```
1  template <ComputationMethod Method>
2  struct _ComputeMeasures<2, 2, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4      static void compute(
5          MakeMeshDataContainer_t<
6              Real,
7              make_custom_integer_sequence_t<unsigned int, 1, 2>
8          >& measures,
9          MeshElements<2, IndexType, Real, Reserve...>& mesh){
10
11         auto& surfaceMeasures = measures.template getDataByDim<2>();
12
13         for (IndexType cellIndex = 0; cellIndex < mesh.getCells().size(); cellIndex++) {
14             const auto& cell = mesh.getCells().at(cellIndex);
15             IndexType tmpEdge = cell.getBoundaryElementIndex();
16
17             Real measure = Real();
18             const Vertex<2,Real>& cellCenter = cell.getCenter();
19             do {
20
21                 auto& edge = mesh.getEdges().at(tmpEdge);
22                 Vertex<2,Real>& a = mesh.getVertices().at(edge.getVertexAIndex());
23                 Vertex<2,Real>& b = mesh.getVertices().at(edge.getVertexBIndex());
24                 double tmp = (cellCenter[0] - a[0]) * (b[1] - a[1]);
25                 tmp -= (cellCenter[1] - a[1]) * (b[0] - a[0]);
26                 measure += 0.5 * fabs(tmp);
27
28                 tmpEdge = mesh.getEdges().at(tmpEdge).getNextBElem(cellIndex);
29             } while (tmpEdge != cell.getBoundaryElementIndex());
30
31             surfaceMeasures.at(cellIndex) = measure;
32         }
33     }
34 };
```

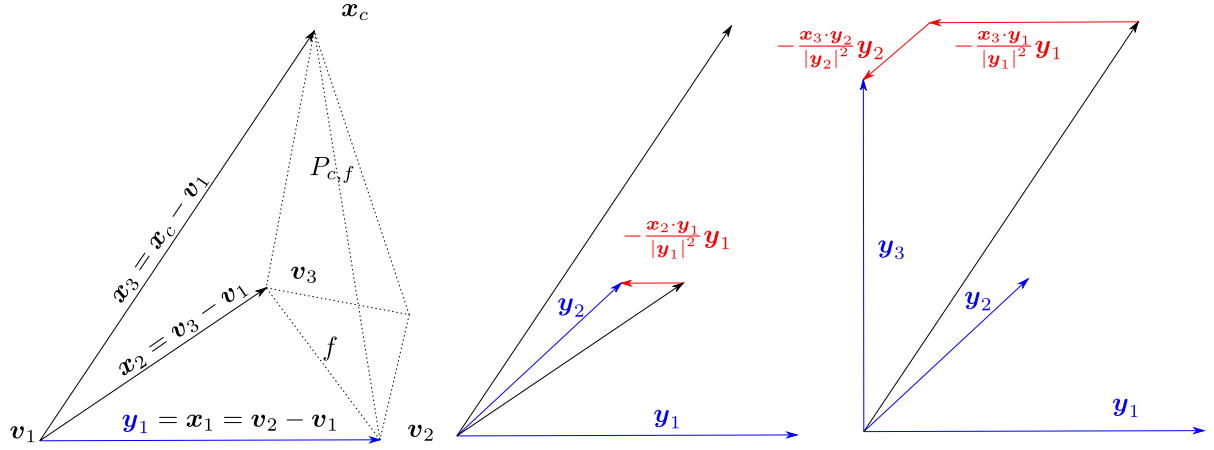



Figure 2.8: An example of calculating $\text{dist}(S_f, \mathbf{x}_c - \mathbf{v}_1)$ to calculate the volume of the pyramid $P_{c,f}$. In terms of equation (2.13), in 3D the element e is the cell c and e' is the face f . The three vertices $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\} \subset N^0(f)$ together with \mathbf{x}_c creates a tetrahedron, which has the same height as $P_{c,f}$. Therefore, it is sufficient to construct the span S_f using three vertices. Note that those vertices must not lie in a line to satisfy the linear independence condition in the Gram-Schmidt process. Finally, after applying the Gram-Schmidt process, the norm of the last vector satisfies $|\mathbf{y}_3| = \text{dist}(S_f, \mathbf{x}_c - \mathbf{v}_1)$. Note that \mathbf{y}_3 must be calculated as the last one, in order for the algorithm to work properly.

The distance is then calculated as $|\mathbf{w} - \mathbf{x}_f|$. For better understanding, see Figure 2.7.

In 3D (and in any higher dimension), this generalizes to applying the Gram-Schmidt process described by the following theorem [24].

Theorem 7. *Let $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ be linearly independent system of vectors in \mathbb{R}^n , $k \leq n$. Then there exists an orthogonal, system $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k$ such that $\mathbf{x}_1 = \mathbf{y}_1$ and*

$$\text{span}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k) = \text{span}(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k).$$

The vectors \mathbf{y}_i are given by the formula

$$\mathbf{y}_i = \mathbf{x}_i - \sum_{j=1}^{i-1} \frac{\mathbf{x}_i \cdot \mathbf{y}_j}{|\mathbf{y}_j|^2} \mathbf{y}_j, \forall i \in \{1, 2, \dots, k\}. \quad (2.12)$$

The result is obtained by utilizing the Gram-Schmidt process on the sequence $(\mathbf{z}_1, \dots, \mathbf{z}_{d-1}, \mathbf{z}_e)$, where

$$\begin{aligned} \mathbf{z}_i &= \mathbf{v}_i - \mathbf{v}_1, \forall i \in \{2, 3, \dots, d-1\}, \\ \mathbf{z}_e &= \mathbf{x}_e - \mathbf{v}_1, \end{aligned}$$

where $e \in \mathcal{F}^d$, $e' \in N^{d-1}(e)$, $\mathbf{v}_i \in N^0(e')$. The $\mathbf{z}_1, \dots, \mathbf{z}_{d-1} \in S_{e'}$ are linearly independent vectors. Note that it is important to process the vector $\mathbf{z}_e = \mathbf{x}_e - \mathbf{v}_1$ as the last one. Finally, the last vector \mathbf{y}_d satisfies

$$|\mathbf{y}_d| = \text{dist}(S_{e'}, \mathbf{x}_e - \mathbf{v}_1). \quad (2.13)$$

This result gives us a way to calculate the volume of any polytope and also provides the calculation of normal vectors to edges and faces (see Section 2.4.7 below). See an example of utilizing the Gram-Schmidt process to calculate the height of a 3D pyramid in Figure 2.8.

The implementation of `_ComputeMeasures` for the case of a 3D mesh has two specializations. The first specialization of `_ComputeMeasures` is for `MeshDimension` equal to 3 and `CurrentDimension` equal to 2, i.e., it handles the calculation of measures of the faces of the

Code listing 2.54 The implementation of the specialization of the class `_ComputeMeasures` for the case of faces in a 3D mesh. For all faces in the mesh, the algorithm loops over the face edges and calculates the area of single triangles which are then summed as the measure of the face, see lines 18-31. Note that the utilized implementation of the `gramSchmidt` function actually normalizes the vectors, therefore the norms are returned in the container passed as the second argument. Otherwise, the information about the distance would be lost. In this case the height of the triangle is stored in `norms[1]`, see line 30. After calculating the measures of the faces, the computation of the measures of the cells is performed.

```

1  template <ComputationMethod Method>
2  struct _ComputeMeasures<2, 3, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4      static void compute(
5          MakeMeshDataContainer_t<
6              Real,
7              make_custom_integer_sequence_t<unsigned int, 1, 3>
8          >& measures,
9          MeshElements<3, IndexType, Real, Reserve...>& mesh){
10
11         auto& surfaceMeasures = measures.template getDataByDim<2>();
12
13         for (IndexType faceIndex = 0; faceIndex < mesh.getFaces().size(); faceIndex++) {
14
15             const auto& face = mesh.template getElements<2>().at(faceIndex);
16             Real measure = Real();
17             const Vertex<3,Real>& faceCenter = face.getCenter();
18             for(auto sube : face.getSubelements()){
19                 const auto& edge = mesh.getEdges().at(sube);
20                 Vertex<3,Real>& a = mesh.getVertices().at(edge.getVertexAIndex());
21                 Vertex<3,Real>& b = mesh.getVertices().at(edge.getVertexBIndex());
22                 // Prepare the vectors to be the G-S process applied on
23                 std::array<Vertex<3,Real>, 2> gsVecs = {b - a, faceCenter - a};
24                 // Prepare the container to store the norms of the processed vectors
25                 std::array<Real, 2> gsNorms = {};
26                 // Apply the Gram-Schmidt algorithm to the vectors
27                 gramSchmidt<2,3,IndexType, Real>(gsVecs, gsNorms);
28                 // Add the measure of the triangle [a,b,face center]
29                 // to the measure of the face
30                 measure += 0.5 * gsNorms[1] * measures.template getDataByDim<1>()[sube];
31             }
32             surfaceMeasures.at(faceIndex) = measure;
33         }
34         _ComputeMeasures<3, 3>::compute(measures, mesh);
35     }
36 };

```

Code listing 2.55 The specialization of `_CalculateMeasures` computing the measures of 3D cells. In this case the algorithm loops over all faces of all cells. Firstly, the center vertex of the face and the vertices of the first edge connected to the face are accessed. Then, the vectors to apply the Gram-Schmidt algorithm are prepared in the `gsVectors` container by subtracting the vertex `a` from the others (see lines 21-37). Let us recall that the vector `cellCenter - a` must be the last vector in the container. Finally, the volume of the pyramid $P_{c,f}$ is calculated by the formula (2.6) and added to the volume of the cell at line 40.

```

1  template <ComputationMethod Method>
2  struct _ComputeMeasures<3, 3, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4      static void compute(
5          MakeMeshDataContainer_t<
6              Real,
7              make_custom_integer_sequence_t<unsigned int, 1, 3>
8          >& measures,
9          MeshElements<3, IndexType, Real, Reserve...>& mesh){
10
11         auto& cellMeasures = measures.template getDataByDim<3>();
12
13         for (IndexType cellIndex = 0; cellIndex < mesh.getCells().size(); cellIndex++) {
14
15             const auto& cell = mesh.getCells().at(cellIndex);
16             IndexType tmpFace = cell.getBoundaryElementIndex();
17             Real measure = Real();
18             const Vertex<3,Real>& cellCenter = cell.getCenter();
19
20             do {
21                 auto& face = mesh.getFaces().at(tmpFace);
22                 auto& firstEdge = mesh.getEdges().at(face.getSubelements()[0]);
23                 // select 3 different vertices
24
25                 Vertex<3,Real>& a = mesh.getVertices().at(firstEdge.getVertexAIndex());
26                 Vertex<3,Real>& b = mesh.getVertices().at(firstEdge.getVertexBIndex());
27
28                 // It is robust to choose the center point to avoid the possibility
29                 // that the three vertices lie in a line
30                 Vertex<3,Real>& c = mesh.getFaces().at(tmpFace).getCenter();
31                 // Prepare the vectors to apply the G-S process on
32                 std::array<Vertex<3,Real>, 3> gsVecs = {b-a, c-a, cellCenter - a};
33                 // Because the vectors are normalized by the gramSchmidt function the norms
34                 // are returned through the second parameter of gramSchmidt function
35                 std::array<Real, 3> gsNorms = {};
36
37                 gramSchmidt<3,3,IndexType, Real>(gsVecs, gsNorms);
38
39                 Real distance = gsNorms[2];
40                 measure += (1.0/3.0) * measures.template getDataByDim<2>().at(tmpFace) *
41                     distance;
42
43                 tmpFace = mesh.getFaces().at(tmpFace).getNextBElem(cellIndex);
44             } while (tmpFace != cell.getBoundaryElementIndex());
45
46             cellMeasures.at(cellIndex) = measure;
47         }
48     }
49 };

```

Code listing 2.56 The wrapper function simplifying the work with the `_ComputeMeasures` class. The function firstly allocates the result container for the measures. Then, the `compute` member function of `_ComputeMeasures` is called. Finally, the result is returned. At line 24, an example of the usage of the function `computeMeasures` is shown. Notice that the user does not need to know the result type at all.

```

1  template <
2      ComputationMethod Method,
3      unsigned int MeshDimension,
4      typename IndexType,
5      typename Real,
6      unsigned int ...Reserve
7  >
8  MakeMeshDataContainer_t<
9      Real,
10     make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>
11 >
12 computeMeasures(MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh){
13     MakeMeshDataContainer_t<
14         Real,
15         make_custom_integer_sequence_t<unsigned int, 1, MeshDimension>
16     > measures(mesh);
17
18     Impl::_ComputeMeasures<1, MeshDimension, Method>::compute(measures, mesh);
19
20     return measures;
21 }
22
23 // ** usage **
24 auto elementMeasures = computeMeasures<METHOD_DEFAULT>(mesh);

```

mesh. In contrast to the calculation in 2D, this calculation does not utilize the equation (2.10), but the generic approach. The realization in GTMesh is presented in Code listing 2.54.

The second specialization of `_ComputeMeasures` performs the computation of cell measures in 3D. In this case, we meet a problem of detecting vertices that does not lie in a line in order to calculate S_f (see equation (2.8)) correctly. This can be done by choosing three suitable vertices lying in the face. In the implementation we chose the vertices of the first connected edge and the center point of the face. The code implementing the described algorithm is shown in Code listing 2.55.

Let us note that the cell measure computation takes advantage of the planarity of the faces. Therefore, the whole face is contained in S_f of dimension 2 and the volume of the pyramid can be calculated at once. However, further in this work we discuss meshes with non-planar faces. This must be considered by the computation of measures. Similarly to the `_ComputeCenters` class, an alternate algorithm for measures computation that takes non-planar faces into account can be selected by means of the `Method` template parameter. More detailed description is in Section 2.7.

The `computeMeasures` Function As we have already noted, the `_ComputeMeasures` class in not designed to be directly used by the user. The reason is that the class `_ComputeMeasures` has more parameters such as `CurrentDimension`, which has to be set to 1 at the start of the function. Moreover, the parameter `measures` has a complicated type and must be already allocated when the member function `compute` is called. The GTMesh library provides the `computeMeasures` function which wraps this functionality and ensures correct usage of the `_ComputeMeasures` class. The user do not need to know the exact return type of the function, the result can be obtained using the keyword `auto`. See the implementation and usage of `computeMeasures` in Code listing 2.56.

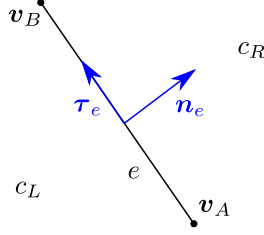


Figure 2.9: The edge is oriented from the vertex v_A to the v_B . τ_e is the tangential vector of the edge and \mathbf{n}_e is the normal vector of the edge. Notice that \mathbf{n}_e points from the left cell c_L to the right adjacent cell c_R . The orientation of an edge in 2D is clockwise with respect to the right cell and counterclockwise with respect to the left cell.

2.4.7 Elements Orientation and the `computeFaceNormals` Function

Another important property of the mesh is the orientation of the elements in space or with respect to other elements. Especially, the normal vectors of faces are important. We denote the normal vector of the face f by \mathbf{n}_f . The vector \mathbf{n}_f is a unit vector and points from the left adjacent cell to the right one. The left and right cells to a face are defined in the utilized data structure, see Section 1.3.3. Note that this is important because some numerical simulations may rely on this. As we already discussed in Section 2.4.6, the problem of calculating the normal vectors was solved when the measure is calculated. So far, the GTMesh library provides the computation of face normal vectors for 2D and 3D mesh only. Nevertheless, there are more ways of face normal calculation, especially in 3D. Firstly, we begin with the with the description of the edge normal calculation in a 2D mesh and then the face normal calculation in the 3D case. Finally, the calculation of orientation of an edge with respect to a face in 3D will be studied.

The computation of the face normal vectors in the GTMesh library is realized by the function `computeFaceNormals` which wraps the implementation in the class `_ComputeNormals` with the public static member function `compute` defined in the `Impl` namespace. The definition of `_ComputeNormals` checks whether the mesh dimension is less or equal to 3 using `static_assert` just as the `_ComputeMeasures` class does.

Edge Normal Vector Calculation in 2D In a 2D mesh, the faces are edges. According to the mesh design, the edges are oriented. The direction of an edge e is from the vertex v_A to vertex v_B (as depicted in Figure 2.9). Thanks to the form of a rotation matrix in 2D, the normal vector calculation can be optimized against the using of the Gram-Schmidt process. The formula for \mathbf{n}_e reads:

$$\mathbf{n}_e = \begin{pmatrix} -(\mathbf{v}_A - \mathbf{v}_B)_2 \\ (\mathbf{v}_A - \mathbf{v}_B)_1 \end{pmatrix} \frac{1}{|\mathbf{v}_A - \mathbf{v}_B|}. \quad (2.14)$$

This calculation of the normal vector does not yet consider whether the normal vector of the edge points from the left cell to the right cell. This shall be done during the mesh construction. The edge shall be oriented clockwise with respect to the right cell and counterclockwise w.r.t. the left one. The correct orientation of the normal vector can be checked by the condition

$$\mathbf{n}_e \cdot (\mathbf{x}_{c_L} - \mathbf{x}_{c_R}) > 0. \quad (2.15)$$

Face Normal Vector Calculation in 3D In a 3D mesh, there are at least two methods of the face normal calculation. Firstly, we discuss the utilization of the vector cross product and why this approach was dismissed in GTMesh. Then, the chosen algorithm utilizing the Gram-Schmidt process will be presented. Let us introduce the geometry setup and the aim of the calculation before the presentation of the methods. The situation for a single face is shown in Figure 2.10. The aim is to calculate the vector \mathbf{n}_f which is orthogonal to the plane defined by the face, i.e. $\text{span}(\tau_{f,1}, \tau_{f,2})$.

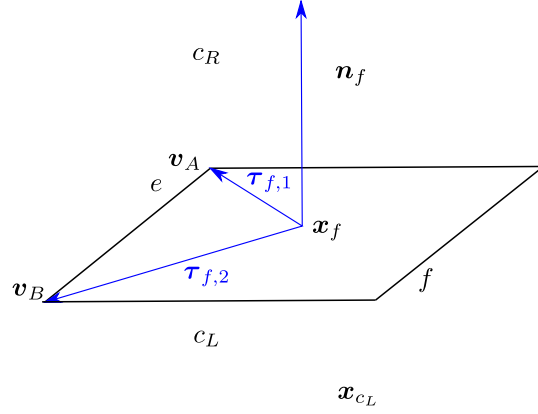


Figure 2.10: The notation on a face f in the context of normal vector calculation with the left cell c_L below and the right cell c_R above. The edge e defined by the vertices v_A, v_B is the first edge of f . \mathbf{x}_{c_L} is the center point of c_L .

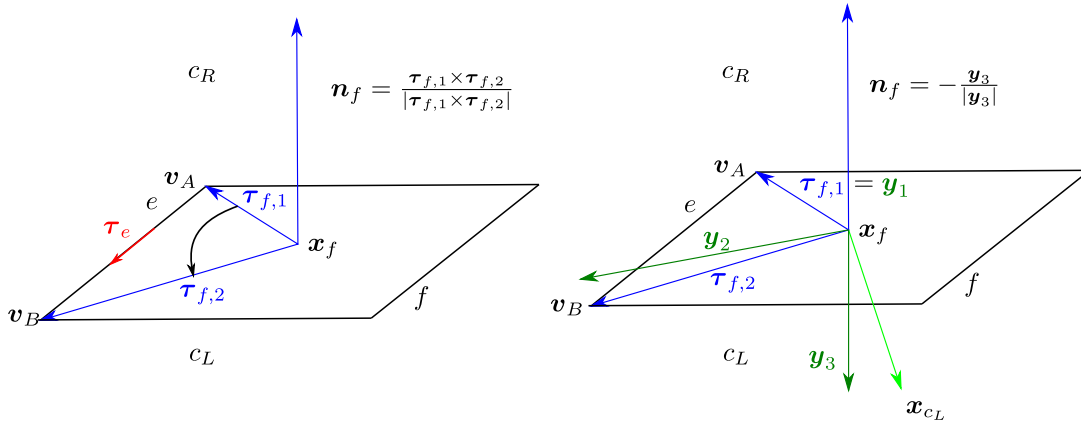


Figure 2.11: Comparison of the two presented approaches of face normal vector calculation. The figure on the left presents the normal vector calculation using the cross product of $\tau_{f,1}$ and $\tau_{f,2}$. The other figure presents the computation of \mathbf{n}_f using the Gram-Schmidt process applied to a triple $(\tau_{f,1}, \tau_{f,2}, \mathbf{x}_f - \mathbf{x}_{c_L})$.

The first method utilizes the vector cross product. The calculation utilizes the vertices of the first edge of the face and the center point of the face. Then, the face normal vector \mathbf{n}'_f is given by the formula

$$\mathbf{n}'_f = \frac{\tau_{f,1} \times \tau_{f,2}}{|\tau_{f,1} \times \tau_{f,2}|} = \frac{(\mathbf{x}_f - \mathbf{v}_A) \times (\mathbf{x}_f - \mathbf{v}_B)}{|(\mathbf{x}_f - \mathbf{v}_A) \times (\mathbf{x}_f - \mathbf{v}_B)|}. \quad (2.16)$$

The advantage of this method of calculation is the simplicity of the formula, therefore the calculation is quick. On the other hand, similarly to the calculation of edge normals in 2D, the formula does not reflect the positions of the left and right cells. Thus, it is necessary to ensure that the direction of the \mathbf{n}'_f is correct.

The second idea does not consider the orientation of the face f . Otherwise, it checks the orientation of the vector \mathbf{n}'_f with the vector $\mathbf{x}_{c_R} - \mathbf{x}_{c_L}$. The formula for the correction then reads

$$\mathbf{n}_f = \begin{cases} \mathbf{n}'_f & \mathbf{n}'_f \cdot (\mathbf{x}_{c_L} - \mathbf{x}_{c_R}) > 0, \\ -\mathbf{n}'_f & \text{otherwise.} \end{cases} \quad (2.17)$$

In spite of the correct result, it was decided not to utilize this solution, because it is limited to 3D meshes only.

Code listing 2.57 The specialization of `_ComputeNormals` for calculation of face normal vectors in 3D meshes. The member function `compute` has two parameters. The first is a `MeshDataContainer` `normals` the result will be stored into. The second is the mesh which the algorithm will be applied to. Then, for each face in the mesh, the left adjacent cell is obtained. If the left cell index is invalid or refers to a boundary cell, the right cell is chosen instead. The condition at line 10 presents how to detect the boundary cell index. The variable `vectorSign` signalizes whether the resulting vector should be reversed. Then, the normal vector is computed by the Gram-Schmidt process and stored into `normals` at the position of the currently processed face.

```

1  template <ComputationMethod Method>
2  struct _ComputeNormals<3, Method>{
3      template <typename IndexType, typename Real, unsigned int ...Reserve>
4      static void compute(MeshDataContainer<Vector<3, Real>, 2>& normals,
5                          const MeshElements<3, IndexType, Real, Reserve...>& mesh){
6          for (auto& face : mesh.getFaces()) {
7              IndexType cellIndex = face.getCellLeftIndex();
8              double vectorSign = 1.0;
9              // If the left cell is boundary, then choose the right cell
10             if (isInvalidIndex( cellIndex ) || isBoundaryIndex( cellIndex )) {
11                 vectorSign = -1.0;
12                 cellIndex = face.getCellRightIndex();
13             }
14
15             const Vertex<3,Real>& cellCenter = mesh.getCells().at(cellIndex).getCenter();
16             // Get 3 vertices that do not lie in a line
17             IndexType vA = mesh.getEdges().at(face.getSubelements()[0]).getVertexAIndex();
18             IndexType vB = mesh.getEdges().at(face.getSubelements()[0]).getVertexBIndex();
19
20             const Vertex<3,Real>& a = mesh.getVertices().at(vA);
21             const Vertex<3,Real>& b = mesh.getVertices().at(vB);
22             // The center is probably not in line with a and b
23             const Vertex<3,Real>& c = face.getCenter();
24             // Create the system of vectors
25             std::array<Vertex<3,Real>, 3> gsVecs = {c - a, c - b, c - cellCenter};
26             std::array<Real, 3> gsNorms = {};
27
28             gramSchmidt<3,3,IndexType, Real>(gsVecs, gsNorms);
29
30             normals[face] = vectorSign * gsVecs[2];
31         }
32     }
33 };
34

```

The implemented solution of the face normal calculation utilizes the Gram-Schmidt process on the vectors $\boldsymbol{\tau}_{f,1}$, $\boldsymbol{\tau}_{f,2}$ and $\boldsymbol{x}_f - \boldsymbol{x}_{c_L}$. The result is obtained as

$$\boldsymbol{n}_f = -\frac{\boldsymbol{y}_3}{|\boldsymbol{y}_3|}, \quad (2.18)$$

where \boldsymbol{y}_3 is the last vector obtained from the Gram-Schmidt process. If the face f does not have any left cell attached, i.e., it lies at the boundary of the mesh, the last vector passed to the Gram-Schmidt process must be changed to $\boldsymbol{x}_{c_R} - \boldsymbol{x}_f$. The advantage of this approach is the possibility to easily extend the algorithm to a generic dimension and the fact that it automatically respects the direction from the left to the right cell. A comparison of the two proposed methods is shown in Figure 2.11.

The implementation of the `_ComputeNormals` calculating the face normal vectors in 3D mesh is shown in Code listing 2.57. Notice the similarity to the corresponding algorithm calculating the measures (Code listing 2.55). Unlike the measure calculation, the member function `compute` of the `_ComputeNormals` class loops over faces only.

Face Normal Vector Calculation in a Generic Dimension In the end, let us briefly comment the generalization of the previous algorithm into a generic dimension. The generic algorithm require two changes in implementation compared to the 3D case. The first is connected to the suitable choice of vertices defining V_f introduced in (2.6). In order to minimize the possibility that three or more chosen vertices lie in a line, the centers of sub-elements shall be chosen except for edges. Then, the formula for V_f reads

$$V_f = \text{span}(\mathbf{v}_A, \mathbf{v}_B, \mathbf{x}_{e^2}, \dots, \mathbf{x}_{e^{d_{\mathcal{F}}-2}}) - \mathbf{x}_f, \quad (2.19)$$

where $e^n \in N^n(f)$, $n \in \mathbb{N}$, $1 < n < d_{\mathcal{F}} - 1$, i.e. e^n are sub-elements of f with dimension n . Furthermore, \mathbf{v}_A and \mathbf{v}_B are the vertices of the first accessed edge. To obtain such sequence of vertices for each face, a special function must be developed. The second change can be demonstrated on a 4D mesh. The requested vertices are $\{\mathbf{v}_A, \mathbf{v}_B, \mathbf{x}_{e^2}\}$, but in the 4D case, only elements of dimension 3 and 4 are considered as computationally significant. Therefore, the mesh does not provide the centers of elements with dimension 2. Thus, the centers would have to be passed to the function as a separate parameter. Similar changes are to be done in the case of measure computation in a generic dimension.

2.5 Mesh Import and Export

In the previous Section 2.4, the functions calculating the mesh properties were discussed. In order to conveniently work with the mesh, it is necessary to be able to load the mesh from a file. Therefore, this section aims to describe the problems connected to input and output of the mesh. Currently, GTMesh provides the import and export of the unstructured mesh in the VTK (2D and 3D) and FPMA (3D) formats[23, 2]. The formats and the import and export process will be described below.

In order to provide compatibility of GTMesh with various mesh formats, the generic types of cells (describing the cell shape) and the `MeshReader` and `MeshWriter` abstract classes were developed (see Code listing 2.58). `MeshReader` has the pure virtual member function `getCellTypes` returning the types of the loaded cells.

Let us begin with the description of the VTK format for unstructured meshes.

2.5.1 VTK format

In the context of unstructured meshes, the VTK data format is designed to store both the mesh description and the mapped data [23]. Thus, we will discuss the methods of importing and exporting the data. In VTK, the cells are represented by vertices which construct the cells. A cell is defined by a sequence of vertices and a type representing the cell shape. The structure of the VTK file is presented in Code listing 2.59. This format utilizes only several primitive types. Therefore, the 3D meshes that can be described by this format are limited. We start with the description of the import and export of the mesh, then we will focus on the mapped data.

Mesh Input

The VTK format is able to store 2D and 3D meshes. Let us begin with the description of the import of a 2D mesh, then the 3D case will be discussed.

In 2D, the geometrical primitives (i.e., cell types) representable in VTK are: triangle, square, and polygon. All those primitives have the same representation in the VTK file. The cells are described by a sequence of vertices ordered counterclockwise along the boundary of the cell. Therefore, all the 2D primitives can be loaded by the same algorithm regardless of the cell type.

The algorithm loading a 2D mesh is the following:

Code listing 2.58 The definitions of the basic constants and base classes for mesh readers and writers.

```
1  template <unsigned int MeshDimension>
2  struct MeshNativeType{};
3
4  // Element types of GTMesh in 2D mesh
5  template<>
6  struct MeshNativeType<2>{
7      enum ElementType{
8          LINE = 100,
9          TRIANGLE = 200,
10         QUAD,
11         POLYGON
12     };
13 };
14
15 // Element types of GTMesh in 3D mesh
16 template<>
17 struct MeshNativeType<3>{
18     enum ElementType{
19         TETRA = 300,
20         HEXAHEDRON,
21         WEDGE,
22         PYRAMID,
23         N_PYRAMID,
24         POLYHEDRON
25     };
26 };
27
28 // The base class of MeshReaders
29 template<unsigned int MeshDimension>
30 class MeshReader{
31 public:
32     using type = MeshNativeType<MeshDimension>;
33     // Pure virtual method returning the types of loaded cells
34     virtual
35     MeshDataContainer<typename type::ElementType, MeshDimension>
36     getCellTypes() const = 0;
37 };
38
39 // The base class of MeshWriters
40 template<unsigned int MeshDimension>
41 class MeshWriter{
42 public:
43     // Definition of the virtual destructor enforces polymorphism of
44     // the derived classes of MeshWriter
45     virtual ~MeshWriter() = default;
46     using type = MeshNativeType<MeshDimension>;
47     // Method calculating returning the signature
48     template<typename IndexType, typename Real, unsigned int ...Reserve>
49     static size_t computeHash(
50         MeshElements<MeshDimension, IndexType, Real, Reserve...>& mesh){
51         // Do not trust the user and recalculate the signature
52         return mesh.updateSignature();
53     }
54 };
```

Code listing 2.59 An example of the VTK format. The vertices are given by 3 coordinates (even in 2D case). The cells are defined by a sequence of indexes of vertices and by a type which determines the shape of the cell, i.e., which vertices make up an edge or a face.

```

1 # vtk DataFile Version 2.0
2 data name
3 ASCII
4 DATASET UNSTRUCTURED_GRID
5 POINTS n double
6 points...
7
8 CELLS m 1
9 cells...
10
11 CELL_TYPES m
12 types...
13
14 POINT_DATA n
15 ...
16
17 CELL_DATA m
18 SCALARS quantity_name double 1
19 LOOKUP_TABLE default
20 ...
21
22 VECTORS vector_quantitiy_name double
23 ...

```

1. Store all vertices in the data structure.
2. For each cell:
 - (a) Read the vertices defining the cell.
 - (b) Pair the vertices that define an edge.
 - (c) Check the list of free edges (edges with only one adjacent cell at the moment) whether an edge defined by the currently processed vertex pair is already created.
 - i. If the edge is not found create it and store it in the respective container, and add it to the list of free edges.
 - (d) Connect the edges to form the currently constructed cell.
 - (e) Set up the cell's boundary edge index.

In GTMesh, this functionality is provided by the class `VTKMeshReader`, which inherits the base class `MeshReader`. The `VTKMeshReader` class has the member function `loadFromStream`, which is given an input stream and a mesh to be loaded. The most important part of `VTKMeshReader` is the private member function `loadCells`, presented in Code listing 2.60. This function performs step 2 of the above algorithm (see Code listing 2.60). To detect whether the currently loaded edge is already created, therefore the algorithm utilizes `std::unordered_map` to store the already constructed edges. An edge is uniquely given by a pair of vertex indexes irrespective their order. The map stores key-value pairs where the key is a string of the vertex indexes in increasing order and the corresponding value if the edge index within `MeshElements`. Each vertex pair of the currently loaded cell is searched for in the map. Depending on the search result, either an existing edge is retrieved from the map or a new one is created. Finally, the cell is connected to this edge.

When the cells are loaded, the types of the cells are stored into the `VTKMeshReader` class in the `cellTypes` attribute. This attribute is returned by the abstract member function `getCellType`. This way the cell types are still preserved in GTMesh, although the types are not stored in the

Code listing 2.60 The realization of the algorithm loading the 2D cells. The algorithm needs to search whether the currently loaded edge element is already present in the mesh. Therefore, the hash table via `std::unordered_map` is utilized to find the edge in the mesh quickly.

```

1 // ** class VTKMeshReader<2> : public MeshReader<2> **
2 template<typename IndexType, typename Real, unsigned int ...Reserve>
3 void loadCells(std::istream& ist, MeshElements<2, IndexType, Real, Reserve...>& mesh){
4     // Map of the created edges
5     std::unordered_map<std::string, IndexType> edges;
6
7     IndexType numCells;
8     ist >> numCells;
9     mesh.getCells().resize(numCells);
10    // Skip the total number of written elements (required by VTK)
11    ist.ignore(50, '\n');
12    for (IndexType cellIndex = 0; cellIndex < numCells; cellIndex++) {
13        // Setup new cell
14        mesh.getCells().at(cellIndex).setIndex(cellIndex);
15        // Read the number of vertices defining the cell
16        IndexType numVert;
17        ist >> numVert;
18        // Load the indexes of the vertices defining the cell
19        std::vector<IndexType> vertices(numVert);
20        for(IndexType j = 0; j < numVert; j++){
21            ist >> vertices.at(j);
22        }
23        // Process the loaded vertices and construct one cell
24        IndexType prevEdge = INVALID_INDEX(IndexType);
25        for(IndexType j = 0; j < numVert; j++){
26            // Setup the vertices of the new edge
27            IndexType iA = vertices.at(j), iB = vertices.at((j+1)%numVert);
28            // Prepare the string key into edges map
29            std::string edgeKey = iA < iB ? std::to_string(iA) + ";" + std::to_string(iB) :
30                                     std::to_string(iB) + ";" + std::to_string(iA);
31
32            // Find the edge in the map
33            auto edgeIt = edges.find(edgeKey);
34
35            IndexType edgeIndex = IndexType();
36
37            // If the edge is not already constructed, construct new
38            if (edgeIt == edges.end()){
39                edgeIndex = mesh.getEdges().size();
40                mesh.getEdges().push_back({});
41                mesh.getEdges().at(edgeIndex).setVertexAIndex(iA);
42                mesh.getEdges().at(edgeIndex).setVertexBIndex(iB);
43                mesh.getEdges().at(edgeIndex).setIndex(edgeIndex);
44                mesh.getEdges().at(edgeIndex).setCellLeftIndex(cellIndex);
45                edges[edgeKey] = edgeIndex;
46            } else {
47                // If the edge has been found in edges map, connect the cell to it
48                edgeIndex = edgeIt->second;
49                mesh.getEdges().at(edgeIt->second).setCellRightIndex(cellIndex);
50            }
51            // Connect the edges as they were loaded
52            if (prevEdge != INVALID_INDEX(IndexType)){
53                mesh.getEdges().at(prevEdge).setNextBElem(edgeIndex, cellIndex);
54            }
55            // Setup the reference of the cell to its boundary element
56            if (j == 0){
57                mesh.getCells().at(cellIndex).setBoundaryElementIndex(edgeIndex);
58            }
59            // Connect the first and the last edge
60            if (j == numVert - 1) {
61                mesh.getEdges().at(edgeIndex).setNextBElem(
62                    mesh.getCells().at(cellIndex).getBoundaryElementIndex(),
63                    cellIndex
64                );
65            }
66            prevEdge = edgeIndex;
67        }
68    }
69 }

```

Code listing 2.61 This listing presents the definition of construction of a hexahedron in the class `VTKMeshReader`. Firstly, the pairs of vertex indexes that create edges in the hexahedron are defined, e.g., the first edge is defined by the first and the second vertex of the sequence. Secondly, the sequences of edges that define the faces are listed, e.g., the first face of a hexahedron is created by the first four edges.

```

1  template<>
2  class VTKMeshReader<3> : public MeshReader<3>{
3      // Map describing which vertices connects to edges
4      // and which edges connects to faces
5      std::map<
6          int,
7          std::pair<
8              std::vector<std::array<int,2>>,
9              std::vector<std::vector<int>>>
10     > TypeEdgesFaces{
11     {8, { // Hexahedron has 8 vertices
12         { // Edges (first)
13             {0,1},{1,2},{2,3},{3,0},{0,4},{1,5},{2,6},{3,7},{4,5},{5,6},{6,7},{7,4}
14         }, { // Faces (second)
15             {0,1,2,3},
16             {4,0,5,8},
17             {5,1,6,9},
18             {6,2,7,10},
19             {7,3,4,11},
20             {8,9,10,11}
21         }
22     }
23     }, // ... Description of the other primitives
24     };
25     // ... Methods loading and constructing the mesh
26 };

```

mesh itself. Note that the cell types were not needed in order to load the cells, because the loading algorithm does not depend on them.

In the 3D case, the problem gains on complexity, nevertheless it is fundamentally the same. Similarly to the construction of edges in 2D, the problem of element connections applies to both edges and faces in 3D. Moreover, the topology of the geometrical primitives is more complex in 3D. The cell primitives in 3D are: tetrahedron, voxel, hexahedron, wedge, and pyramid. In contrast to the 2D case, those primitives have different description by the vertices, see [23]. The main pieces information we need to construct a 3D cell element are:

1. pairs of vertices defining edges in the cell,
2. sequences of edges defining the faces of the cell.

When we have this information, it is possible to develop a generic algorithm which loads any cell type. Similarly to the 2D case, the cell types are not important for GTMesh. The reason is that the 3D primitives differ in the number of vertices, therefore the type can be deduced during loading (of) the cell vertices. The only coincidence is between the voxel and hexahedron, but GTMesh does not distinguish between those two objects. A voxel is not supported in GTMesh as it is a special case of a hexahedron. The example of the list defining the edges and faces of hexahedron from the sequence of vertices is shown in Code listing 2.61.

Mesh Output The next task is the mesh output, i.e., export of the mesh topology and geometry in the VTK format. GTMesh again provides the export of 2D and 3D meshes. The export of the mesh into VTK is realized by the member function `writeToStream` of the class `VTKMeshWriter`. The `VTKMeshWriter` class has three template parameters:

1. `MeshDimension` (`unsigned int`): dimension of the mesh to be exported,

2. `IndexType (typename)`: type of indexes in the mesh,
3. `Real (typename)`: type of the coordinates of the vertices.

The class `VTKMeshWriter` has two specializations for `MeshDimension` equal to 2 and 3. Both specializations utilize mesh indexing, i.e., first they prepare the data needed for the export and then export the mesh. Because the preparation of the data to export the mesh is a very demanding task, `VTKMeshWriter` stores the indexed data, and it remembers the signature of the indexed mesh. When the same mesh is to be exported again, the step of mesh indexing can be skipped and the mesh is exported quickly (together with the simulation data that may have changed). The function `writeToStream` in both specialization has the following parameters:

4. `ost (std::ostream)`: the stream the mesh will be exported into,
5. `mesh (MeshElements)`: the mesh to be exported,
6. `cellTypes (MeshDataContainer<ElementType, MeshDimension>)`: cell types representing the geometry of cell elements.

If the given mesh has a different signature from the stored one, the mesh is indexed. For the description of the signature of the mesh, see Section 2.5.3. Then, the `writeToStream` member function exports the prepared information in VTK format. Moreover, the `VTKMeshWriter` class provides the `writeHeader` member function, that writes a VTK header and the name of the exported mesh and data.

The 2D case is simpler to implement than the 3D one, so we start with the 2D mesh export. Luckily, despite usage of geometrical primitives, the VTK format is able to describe an arbitrary topology in 2D. The only task connected to exporting cell elements in 2D mesh is to correctly order the vertices of the cell, i.e., the vertices must be ordered counterclockwise as they circle the cell [23]. This can be achieved by the following algorithm:

1. Access the cell boundary edge.
2. Export the index of the vertex v_B if the edge is left to the cell and v_A otherwise. Store the indexes of the exported vertex and edge.
3. In a loop, search for the other edge which has the same vertex as was last exported.
4. Write the other vertex of the edge and store the indexes of the exported vertex and edge.
5. Repeat steps 3 and 4 until all vertices adjacent to the given cell are exported.

The above algorithm is generic because it does not rely on the order in which the edges are chained. The obtained sequences of vertex indexes are then stored in the attribute `cellVert`. When the mesh is to be exported, the container `cellVert` is used because it contains the prepared sequences of indexes of vertices defining the corresponding cells.

In 3D, the topology of the mesh described by the VTK format is limited. Therefore, the indexing of the elements depends on the cell types. We will demonstrate the problem of cell indexing on the examples of a pyramid, wedge, and polyhedral cells.

In terms of VTK, a pyramid has a square base. Therefore, if a pyramid cell is to be indexed, the base must be detected at first. The base is the only face of the cell with four vertices. The vertices of this face are then indexed. The correct order of the vertices is ensured by utilizing a similar algorithm to 2D cell indexing. Then, the last top vertex is to be found; this is achieved by searching which of the vertices of the cell is missing in the indexed vertices. The top vertex is then appended to the list of vertices. The export of tetrahedron is the same as the pyramid export, with the exception that any face of the tetrahedron can be considered as the base one.

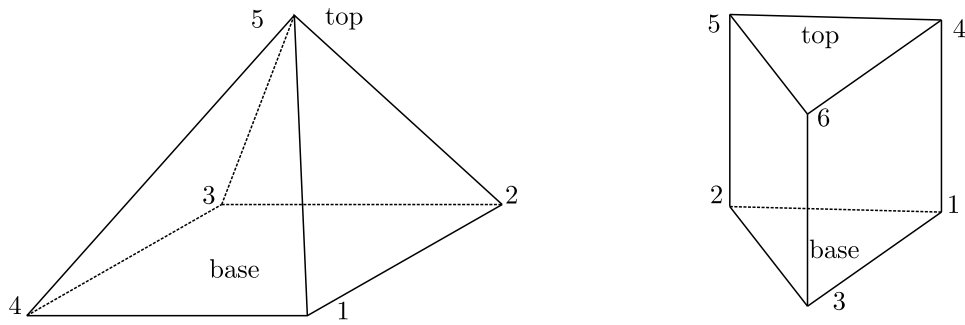


Figure 2.12: The orders of the vertices in the list describing a pyramid and a wedge in VTK unstructured mesh format.

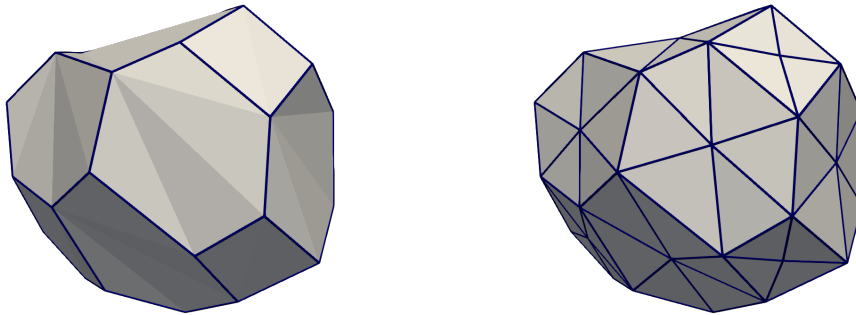


Figure 2.13: An example of the tessellation of polyhedrons. On the left, an original polyhedron is shown. On the right, the polyhedron is divided into tetrahedrons which base triangle lies in the face and the top is the cell center.

The VTK starts the description of the wedge from one of the faces which has three edges. Therefore, one of the faces with three edges must be found first. Then the vertices of this face are put in the list. Then the rest of the vertices are found by searching over the cell edges. The next vertex is found as the one that neighbors with the first indexed one over an edge which is not present in the index yet. This is repeated for all the vertices of the base face. The hexahedron is indexed in the same way, the only difference is that any face of the hexahedron can be the base one.

The last type of the cell is a generic polyhedron. As we already mentioned, the VTK format is not able to store such generic element. Therefore, for the purposes of the export, the polyhedral cells are divided into tetrahedrons. The tetrahedrons are defined by four vertices: \mathbf{x}_c , \mathbf{x}_f , \mathbf{v}_A , \mathbf{v}_B . Such tetrahedron is defined for all edges of all faces of the polyhedral cells. The problem of exporting the polyhedral cells comes together with adding new vertices into the exported mesh. The new vertices are stored in `std::vector` and similarly to mesh indexing, a map is utilized for easier detection whether a vertex have been already added into the mesh. Moreover, the exported mesh has more cells than the one stored in `MeshElements`. For example, if the mesh is made of cells with 10 faces and each face has 10 edges, then the exported mesh has 100 times more cells. In order to map the data from the original mesh onto the exported one, the `VTKMeshWriter` has an attribute named `backwardCellIndexMapping`. This attribute maps the exported cells to the real ones and is utilized by the class `VTKMeshDataWriter` described below.

Data I/O

It was already mentioned that the VTK format is able to store data mapped to the mesh (see Code listing 2.59). GTMesh provides the classes `VTKMeshDataReader` and `VTKMeshDataWriter`

utilizing the concept of class traits (see Section 3). Let us first describe the output and then the input of the data.

The `VTKMeshDataWriter` class is designed to export any data stored in `MeshDataContainer` or `DataContainer` which maps data to the mesh. The export is done using the public static member function `writeToStream`. This function accepts the following arguments:

1. `ost` (`std::ostream`): a stream the data will be exported to,
2. `data` (`MeshDataContainer` or `DataContainer`): data to be exported,
3. `writer` (`VTKMeshWriter`): a writer that exported the mesh, (contains information about the tessellation of polyhedrons).

If the a `DataContainer` is passed as the data parameter, then the contained type must have `DefaultIOTraits` and the attributes of the structure are exported to the VTK file. On the other hand, if the `MeshDataContainer` is passed, the `writeToStream` function firstly searches for the exportable data, i.e., data allocated to cells and with `DefaultIOTraits` defined (see Section 3.3). Then all suitable data in `MeshDataContainer` are exported. If the data does not contain any suitable data, an error occurs. Moreover, the user is told to define default IO class traits (see Section 3.3.1) for the data. The concept of class traits is described in Chapter 3. The purpose of the last parameter `writer` is to ensure the correct mapping of the exported data. This is because the tessellation of polyhedrons into tetrahedrons and their correspondence stored in `backwardCellMapping` must be known. The value mapped to the polyhedron is to be repeated for each tetrahedron separately. The example of working with the `VTKMeshDataWriter` is presented in Code listing 2.62.

Utilizing the concept of class traits, it was possible to create a general method for loading the data. This functionality is provided by the class `VTKMeshDataReader` with the public static member function `readFromStream`. The arguments of the `readFromStream` function are:

1. `ist` (`std::istream`): a stream the data will be read from,
2. `data` (`MeshDataContainer` or `DataContainer`): container the data will be stored into.

Similarly to `VTKMeshDataWriter`, when `MeshDataContainer` is passed, the function `readFromStream` searches for data with `DefaultIOTraits` (see Chapter 3). The `readFromStream` function matches the name labels of data in `DataContainer` or `MeshDataContainer` defined in the class traits with the names of attributes in the data in the VTK file and fills the container with the data. For an example how to read data from a VTK file, see Code listing 2.62. Finally, let us note that the `readFromStream` requires the stream to be opened in binary mode. If the file is opened in the text mode, the `tellg` and `seekg` functions may not work properly depending on on the EOL byte sequence (LF vs. CR vs. CR/LF) used by the system.

2.5.2 FPMA format

The second supported mesh format by the `GTMesh` library is the FPMA format native for AVL FIRE™ [2]. The mesh reader and writer are implemented for 3D meshes only. This format is based on elements connections, i.e., the faces are defined by a sequence of vertices and cells are defined by a list of connected faces. Therefore, there are no restrictions on the topology of the stored mesh and this format does not utilize any cell primitives. Moreover, the FPMA mesh representation is similar to the one utilized in `MeshElements`. Therefore, it is easier to work with and does not require such complex algorithms. The algorithms for input and output of the faces are the same as the corresponding algorithms for 2D cells and the VTK format. As cells are defined directly by the list of the faces, the algorithms constructing and indexing cells are trivial.

Code listing 2.62 A demonstration how to import and export the mesh and the mapped data. At first, the data structure `CellData` to be mapped to the mesh is defined. Then, the default Traits for the `CellData` attributes is defined at line 6 (see Chapter 3) so that the I/O operations accept the mesh to be exported. As the types of cells are not known, we set them as polyhedrons. Then, the mesh is exported to the file `mesh.vtk`. After exporting the mesh, the function calculates the coloring of the cells with respect to connections over vertices and stores the result together with the cell centers in the `meshData` container. `meshData` is then exported to `mesh.vtk` according to the exported (tessellated) mesh. Finally, the tessellated mesh and the mapped data, i.e., centers and coloring of the original mesh, are loaded from the file.

```

1  struct CellData {
2      unsigned int color;
3      Vertex<3, double> center;
4  };
5  // Create default Traits for the class CellData
6  MAKE_ATTRIBUTE_TRAIT(CellData, center, color);
7  // Example of import and export of mesh and data in VTK unstructured mesh file format
8  void foo(UnstructuredMesh<3, size_t, double, 6>& mesh) {
9      // Export the mesh first
10     // In order to enforce the mesh tessellation, prepare the cell types as polyhedrons
11     // the correct cell types can be obtained here: reader.getCellTypes()
12     MeshDataContainer<MeshNativeType<3>::ElementType, 3> cellTypes;
13     cellTypes(mesh, MeshNativeType<3>::POLYHEDRON);
14
15     // Write the mesh to file
16     VTKMeshWriter<3, size_t, double> writer1;
17     std::ofstream out3D("mesh.vtk");
18     writer1.writeHeader(out3D, "test_data");
19     // Write the tessellated mesh
20     writer1.writeToStream(out3D, mesh, cellTypes);
21     // This would export the original mesh
22     // writer1.writeToStream(out3D, mesh, reader.getCellTypes());
23
24     // Calculate mesh properties and store them into meshData
25     MeshDataContainer<CellData, 3> meshData(mesh);
26     auto colors1 = ColorMesh<3, 0>::color(mesh);
27     for(auto& cell : mesh.getCells()){
28         meshData.at(cell).color = colors1.at(cell);
29         meshData.at(cell).center = cell.getCenter();
30     }
31     // Export the data mapped to the mesh
32     VTKMeshDataWriter<3>::writeToStream(out3D, meshData, writer);
33     out3D.close();
34
35     // Load the mesh from file
36     ifstream in3D("mesh.vtk", std::ios::binary); // VTKMeshDataReader requires binary mode
37     VTKMeshReader<3> reader;
38     reader.loadFromStream(in3D, mesh);
39     mesh.initializeCenters();
40
41     // Read the exported data from the mesh file
42     MeshDataContainer<CellData, 3> meshDataIn(mesh);
43     VTKMeshDataReader<3, size_t>::readData(in3D, meshDataIn);
44     in3D.close();
45 }

```


Code listing 2.63 Example of unit cube written in the FPMA format. At first the number of vertices of the mesh is written. Then, the vertices coordinates follow. Next, the faces elements are described. The faces are stored as a list of vertices starting by a number of vertices defining the particular face. The cells are defined by a list of indexes referring to adjacent faces. Again, the list begins with the number of faces defining a single cell. Finally, the last number defines number of labels. The labels of special sections are not supported in GTMesh yet.

```

1 8 # number of vertices
2 0 0 0 # coordinates
3 1 0 0
4 0 1 0
5 1 1 0
6 0 0 1
7 1 0 1
8 0 1 1
9 1 1 1
10
11 6 # number of faces
12 4 1 0 2 3 # number of elements and elements list
13 4 4 0 1 5
14 4 5 1 3 7
15 4 7 3 2 6
16 4 6 2 0 4
17 4 5 4 6 7
18 1 # number of cells
19 6 0 1 2 3 4 5
20 1 # 1: number of special selections, here there is one selection named by boudary_face
21 boundary_face # name of the section
22 3 # code for boundary faces
23 6 # number of faces
24 0 1 2 3 4 5 # and the last line is the face indices.

```

2.5.3 Mesh Signature

In the description of the `MeshElements` class, the attribute `meshSignature` were presented. In this section, the purpose and usage of this attribute will be discussed.

Basically, `meshSignature` reflecting the state of the mesh. The main purpose of this concept is to simply detect whether the mesh has changed. It especially useful in the mesh export. When the mesh has not changed since the last export, it is not necessary to index the mesh again in order to export it. Omitting indexing of the mesh is convenient because it is a very computationally demanding task (see Section 2.5). In the `MeshElements` class, there are two methods working with `meshSignature`. The first is called `getMeshSignature`. This member function returns the value of the signature. The other member function is `updateMeshSignature`. This function calculates the signature of the current state of the mesh and stores it in the `meshSignature` attribute. It also returns the currently calculated value. The member function `updateSignature` shall be called after every change of the mesh.

The signature of the mesh is calculated as a hash of the vectors of mesh elements. The functionality is implemented using the private member class `HashOfMeshElements`, which is utilized by the member function `updateMeshSignature`. The member function `hash` of the class `HashOfMeshElements` utilizes `std::hash` and hashes the vector of elements. However, the standard template library does not provide a specialization of `std::hash` for containers. The only similar class is `std::string`. Therefore, the simplest way to hash a vector as a sequence of bytes in memory is to copy the vector into a string, then `std::hash` can be applied. This has to be done in standards prior to C++17. Since C++17, there is a non-owning container named `std::string_view`. This structure has only a pointer into memory and the length of the data array. As the `std::hash` function is overloaded for this type, the function hashing the mesh can utilize `std::string_view` and avoid copying the data. An example of hashing the vector of cells is in Code listing 2.64.

Code listing 2.64 The implementation of the function computing the hash of cells of the `MeshElements` instance. The hash is calculated as a hash of the binary representation of the vector of cells. This is achieved by copying the data into `std::string` or using `std::string_view` (since C++17) and hashing the string by the `std::hash` function. Since the `HashOfMeshElements` class template is a member of `MeshElements`, it must have one more template parameter `Dummy` preventing explicit specialization. This is because an explicit specialization of the member structure is allowed only in an explicit specialization of the encapsulating structure, see [7].

```

1  /** Inside MeshElements class **
2  template<typename Dummy>
3  struct HashOfMeshElements<Dimension, Dummy>{
4      static size_t hash(MeshElements<Dimension, IndexType, Real, Reserve...>& mesh){
5          // Hash of cells
6  #if __cplusplus <= 201702L // C++14 and older
7              std::hash<std::string> hasher;
8              // Use string as a byte container representing the array
9              std::string tmpString(
10                 reinterpret_cast<char*>(mesh.getCells().data()),
11                 mesh.getCells().size() * sizeof (
12                     MeshElements<Dimension, IndexType, Real, Reserve...>::Cell
13                 )
14             );
15             size_t cHash = hasher(tmpString);
16
17 #else // C++17 and later
18             std::hash<std::string_view> hasher;
19             // Use string_view as a byte representation of the vector
20             std::string_view vectorView(
21                 reinterpret_cast<char*>(mesh.getCells().data()),
22                 mesh.getCells().size() * sizeof (
23                     MeshElements<Dimension, IndexType, Real, Reserve...>::Cell
24                 )
25             );
26
27             size_t cHash = hasher(vectorView);
28 #endif
29             // Combine the hash with the hash of lower dimension
30             return cHash ^ HashOfMeshElements<Dimension -1>::hash(mesh);
31     }
32 };

```

2.6 UnstructuredMesh as the MeshElements Wrapper

As described in the project architecture (Section 2.1), the GTMesh aims to provide a single compact class exposing both the mesh structure and the mesh algorithms by means of its member functions, to simplify the work with the mesh. This construction makes the work with the mesh much more convenient. This wrapper class is called `UnstructuredMesh`. It inherits the `MeshElements` class and has no further structure. The only purpose of `UnstructuredMesh` is to provide the algorithms from Sections 2.4 and 2.5 as its public member functions. The provided functions are constructed with as few template parameters as possible because most of the parameters of the mesh functions can be deduced from the setup of `UnstructuredMesh`. For example, the member function `computeElementMeasures` has only one template parameter `Method` because the rest of the parameters of the function `computeMeasures` are deduced. Part of the implementation of the `UnstructuredMesh` wrapper is shown in Code listing 2.65. `UnstructuredMesh` provides the following member functions:

- `initializeCenters <ComputationMethod Method> ()`: Calculates the centers of the cells and faces initializes the centers of computationally significant elements (see Section 2.4.5),
- `computeElementMeasures <ComputationMethod Method> ()`: Computes the measures of elements (see Section 2.4.6),
- `computeFaceNormals <ComputationMethod Method> ()`: Calculates the face normal vectors (see Section 2.4.7),
- `apply <unsigned int StartDim, unsigned int TargetDim, typename Functor > ([IndexType startElementIndex,] const Functor &func)`: Performs loops over mesh elements (see Section 2.4.1),
- `connections <unsigned int StartDim, unsigned int TargetDim, Order ConnectionsOrder> ()`: Determines the connections between mesh elements (see Section 2.4.2),
- `neighborhood <unsigned int StartDim, unsigned int ConnectingDim, unsigned int ConnectedDim, Order ConnectionsOrder> ()`: Determines the neighborhood of mesh elements (see Section 2.4.3),
- `coloring <unsigned int StartDim, unsigned int ConnectingDim, ColoringMethod Method> ([unsigned int seed])`: Creates a proper coloring of the mesh elements (see Section 2.4.4),
- `load (const std::string& filePath)`: Loads the mesh from a file (see Section 2.5),
- `write (const std::string& filePath, [std::unique_ptr<MeshWriter<MeshDimension>> & writer, const MeshReader<MeshDimension>& meshReader, const std::string& dataHeader])`: Writes the mesh into a file (see Section 2.5). This function has many overloads to allow multiple ways of usage.

Most of the functions in `UnstructuredMesh` are implemented by application of the generic global function on `this`, see the presentation of `UnstructuredMesh` definition in Code listing 2.65.

2.7 3D Meshes with Non-planar Faces

During the work on this thesis, we dealt with 3D meshes made of elements such as the one shown in Figure 2.13. From the example element it is obvious that this particular cell has non-planar

Code listing 2.65 The definition of the `UnstructuredMesh` class template and several member functions. The purpose of the `UnstructuredMesh` class is to provide both the mesh structure (`MeshElements`) and mesh algorithms as member functions. `UnstructuredMesh` has no further member attributes and its inner structure is completely inherited from `MeshElements`. The member functions working with the contained mesh are implemented by application of the global functions on itself (`this`), e.g., `computeFaceNormals` at line 22.

```

1 // The UnstructuredMesh class is the wrapper of MeshElements.
2 template <unsigned int MeshDimension,
3         typename IndexType,
4         typename Real,
5         unsigned int ...Reserve>
6 class UnstructuredMesh : public MeshElements<MeshDimension, IndexType, Real, Reserve...>{
7 public:
8     // Initializes the centers of cells and faces of the mesh.
9     template<ComputationMethod Method = ComputationMethod::METHOD_DEFAULT>
10    void initializeCenters(){
11        auto centers = computeCenters<Method>(*this);
12
13        for (auto& face : this->getFaces()){
14            face.setCenter(centers[face]);
15        }
16        for (auto& cell : this->getCells()){
17            cell.setCenter(centers[cell]);
18        }
19    }
20    // Calculates the normal vectors of the faces in the mesh.
21    template<ComputationMethod Method = ComputationMethod::METHOD_DEFAULT>
22    MeshDataContainer<Vector<MeshDimension, Real>, MeshDimension-1> computeFaceNormals() {
23        return computeFaceNormals<Method>(*this);
24    }
25
26    // Applies the passed function func on the elements of target dim.
27    template<unsigned int StartDim, unsigned int TargetDim, typename Functor>
28    void apply(const Functor& func) {
29        return MeshApply<StartDim, TargetDim>::apply(*this, func);
30    }
31
32    // Returns the indexes of connected elements of
33    // dimension TargetDim to elements with dimension StartDim.
34    template<unsigned int StartDim,
35            unsigned int TargetDim,
36            Order ConnectionsOrder = ORDER_ASCEND>
37    MeshDataContainer<std::vector<IndexType>, StartDim> connections() {
38        return MeshConnections<StartDim, TargetDim, ConnectionsOrder>::connections(*this);
39    }
40
41    // ... ** Implementation of other functions **
42 };

```

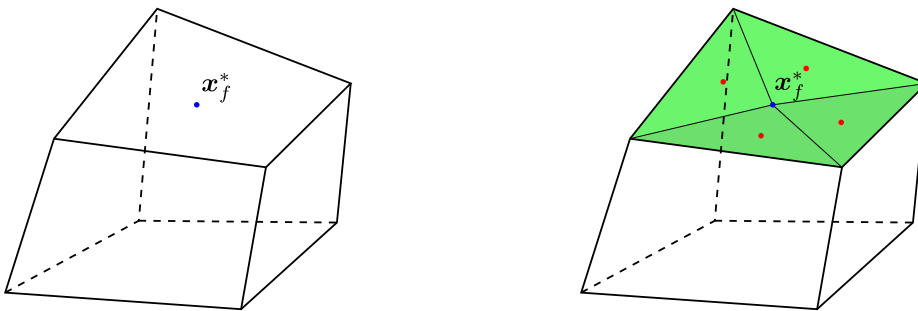


Figure 2.14: When the face of a cell is non-planar, it is tessellated into triangles constructed from the edges and the vertex \mathbf{x}_f^* . The vertex \mathbf{x}_f^* can be, for example, an average of face vertices or given by formula (2.20) (as utilized in [37]). The red points represent the vertices denoted \mathbf{x}_{Δ_e} in equation (2.20).

faces. Such cells are not convex not even star domains. However, there is a way how to correct these qualities which consists in tessellating the faces into triangles [37].

For each face $f \in \mathcal{T}^2$ and an edge $e \in N^1(f)$, a triangle with base e and apex \mathbf{x}_f^* is constructed, where

$$\mathbf{x}_f^* = \frac{\sum_{e \in N^1(f)} m(\Delta_e) \mathbf{x}_{\Delta_e}}{\sum_{e \in N^1(f)} m(\Delta_e)}, \quad (2.20)$$

where Δ_e is a triangle with base e and apex \mathbf{x}_f face centers calculated by `computeCenters`, see Section 2.4.5 (in this case the same as an average of vertices) and \mathbf{x}_{Δ_e} is the center of Δ_e . See the explanatory Figure 2.14.

The process of tessellation leads to planar faces and turns the cells into star domains. From the geometrical point of view, the only difference is that the faces have a richer inner structure which must be respected by the computational algorithms. An example of tessellation of non-planar faces is noticeable in Figure 2.13 presenting the export of generic cells into VTK file format, where a similar tessellation takes place.

In the GTMesh library, there are versions of algorithms such as element center calculation or measure calculation that are able to work with this type of meshes. As was already mentioned in the description of the implementation of the corresponding algorithms (Sections 2.4.5, 2.4.6, 2.4.7), the version of the algorithm is specified by setting the `Method` template parameter. By default, the value of `METHOD_DEFAULT` is assumed. This way, the algorithms consider the mesh to have planar faces. If the `METHOD_TESSELLATED` is set, the algorithms utilizes the tessellated structure of the faces.

Note that if the mesh has planar faces, the results of both versions of the algorithms are the same. One can use the tessellated version when the type of mesh is unknown in advance, but the default version is faster.

Chapter 3

Class Traits

In this chapter, we introduce a concept developed as part of GTMesh and motivated by the automatic export and import of data structures. The goal was to create a tool able to “understand” (to provide a standardized interface to) the data stored in the attributes of classes and structures. Thanks to this tool, it is possible to create functions automatically exporting and importing data stored in such structures to VTK or JSON format, see an example Code listing [2.62](#).

A similar concept was utilized in the work [\[39\]](#), however the concept used was very limited. It was able to handle only 2 data types (scalar and vector quantities) and the reference was to member attributes only. The new concept was partially inspired by marshallers in Java and ThorsSerializer in C++ [\[21, 13\]](#). However, the concept of “understanding” the contents of classes goes beyond the functionality of the mentioned tools.

Motivated by type traits in standard library and ThorsSerializer [\[21\]](#), we decided to name this functionality as *class traits*. A class for which the traits functionality is available will be called a *traited class*. The utilized notation is summarized in [Table 3.1](#). Moreover, it is possible to detect whether a class is traited, hence such classes can be automatically processed. In GTMesh, this concept is mainly provided by the `Traits` class template.

Class traits in GTMesh provide the following functionality:

- enumerate the user-selected data members (attributes) and provide a generic read/write interface to them,
- give names to the enumerated attributes in the form of strings available at run time,
- create an enumerated list of abstract (virtual) attributes with read/write interface implemented using user-defined getter and setter functions.

The exact way how to enable some or all of these particular features will be described in the rest of the chapter. In general, the aim is to provide the given functionality for attributes of an arbitrary type and to make the definition of class traits as simple as possible for the user.

At first, the `Traits` class was able to name the references, but utilized slow virtual member functions to ensure the correct construction. However, as the project evolved the concept of class traits was optimized. The optimization consists in making the calling of the references or functions more direct. The aim was to create code that can be resolved at compile time, so that the access to the members of a traited class is as quick as if the code was explicitly written. Then, the usage of automatically generated member access functions would not slow down the algorithm. Therefore, the class traits might be utilized even in computational algorithms.

In this chapter, the final form of the class traits concept will be discussed. Note that in every iteration of optimizations, the concept has significantly changed. The final form of the class traits consists of the class `MemberAccess` unifying the interface for different types of accessing the members, and the class `Traits` itself. Firstly, the class `MemberAccess` will be described, followed by the construction of `Traits`. Finally, the system of publishing a global instance of `Traits` corresponding to a traited class will be presented.

Term	GTMesh equivalent
class traits	the <code>Traits</code> class template providing access to the attributes of the given class
traited class	the class type the class traits are defined for
default class traits / default traits	specializations of classes <code>DefaultTraits</code> , <code>DefaultIOTraits</code> , <code>DefaultArithmeticTraits</code> publishing an instance of <code>Traits</code> globally
MAKE_TRAIT macros	a group of nine macros simplifying the definition of default class traits, i.e., <code>MAKE_CUSTOM_TRAIT</code> , <code>MAKE_NAMED_ATTRIBUTE_TRAIT</code> , <code>MAKE_ATTRIBUTE_TRAIT</code> and 6 other with “_IO” or “_ARITHMETIC” appended
MAKE_TEMPLATE_TRAIT macros	a group of nine macros simplifying the definition of default class traits for class templates, i.e., <code>MAKE_CUSTOM_TEMPLATE_TRAIT</code> , <code>MAKE_NAMED_ATTRIBUTE_TEMPLATE_TRAIT</code> , <code>MAKE_ATTRIBUTE_TEMPLATE_TRAIT</code> and 6 other with “_IO” or “_ARITHMETIC” appended

Table 3.1: Explanation of terms, used in Chapter 3.

3.1 The MemberAccess Class

The first task in the construction of `Traits` is to create a container facilitating access to the members with a unified interface for various types of references. The aim of the `MemberAccess` is to provide access to a member of an instance of the traited class. The interface consists of two functions `getValue` and `setValue`, as described in detail below. `MemberAccess` must support more ways of accessing the class members, e.g., a pointer to member or a pair of getter and setter functions. To better understand the purpose of `MemberAccess`, see Code listing 3.1.

The definition of `MemberAccess` is very simple, it has a constructor to deduce the template arguments only. Additionally, it utilizes a static assertion to inform the user that an incorrect reference type was passed, see Code listing 3.2. Note that the generic definition of `MemberAccess` must not be used in the context of the specializations.

The specializations of the `MemberAccess` have the following interface:

- `MemberAccess(refType)`: constructor sets the constant reference up,
- `ValueType getValue((const) Class&)`: a “getter” function which returns the value of the data member of the passed object,
- `void setValue(Class&, const ValueType&)`: a “setter” function which sets the given value into the data member,
- `ValueType& getAttr(Class&)`: returns the reference to the member attribute.

According to the type of reference, there are limitations on the interface. For example, if `MemberAccess` utilizes the pair of getter/setter functions, the access is not direct. Therefore, it is not possible to implement the function `getAttr`. Furthermore, when the getter is not set as constant in the class, then the `getValue` function must accept non-constant reference to the traited class only. In order to simply detect whether a particular specialization of `MemberAccess` allows direct access or `getValue` accepts constant reference, the structures `DirectAccess` and `ConstGetAccess` were created. These two structures carry a `constexpr` information which can be utilized to detect the discussed qualities of the `MemberAccess` at compile time. `MemberAccess`

Code listing 3.1 An example of the definition of the `MemberAccess` instances with the given reference. The first two instances of `MemberAccess` provide direct access to the attributes `density` and `momentum` of the class `qties`. The third one (`refVelocity`) accesses the attribute `velocity` provided by the functions `getVelocity` and `setVelocity`. The virtual attribute `velocity` is not stored in directly in `qties`. Instead, its value is calculated from `momentum` and `density`. Then the listing presents how to setup the values of the `qties` instance using the `MemberAccess` instances.

```

1  class qties {
2      double density;
3      Vector<3, double> momentum;
4
5      Vector<3, double> getVelocity(){return momentum/density;}
6      void setVelocity(const Vector<3, double> velocity){momentum = velocity * density;}
7  }qInstance;
8
9  MemberAccess refDensity(&qties::density);
10 MemberAccess refMomentum(&qties::momentum);
11 MemberAccess refVelocity(std::make_pair(&qties::getVelocity, &qties::setVelocity));
12
13 refDensity.getAttr(qInstance) = 3; // Set the density directly
14
15 // When setting the value of velocity, the value of density must be already set
16 refVelocity.setValue(qInstance, {1,2,3}); // Set the velocity using the set function
17
18 refMomentum.getValue(qInstance); // Returns the copy of the momentum, i.e. {3, 6, 9}

```

Code listing 3.2 The definition of the `MemberAccess` class. This definition must not be instantiated, i.e., in every case of use, a specialization must be utilized. If the given reference is not known, the compilation ends with an error.

```

1  template<typename Ref>
2  class MemberAccess{
3      static_assert (!std::is_same<Ref, void>::value,
4                      "The type Ref must be reference to member (ValueType Class::*), or "
5                      "member function or pointer to getter and setter");
6      MemberAccess(Ref);
7  };

```

with the appropriate quality inherits the corresponding structure. For an example of working with `MemberAccess`, see Code listing 3.1. The implementation details of the three most important ways of accesses are discussed below.

The first specialization handles the pointer to a member attribute of the traited class. In this specialization, the reference has the type `ValueType Class::*` (pointer to member), where the `ValueType` is the type of the member and `Class` is the type of the traited class. The value of the pointer to member is set in the constructor of `MemberAccess`. Using the pointers to members, the data are obtained by the expression at line 22 in Code listing 3.3. This type of reference has both the qualities (`DirectApproach` and `ConstGetApproach`).

The next important possibility of access to members utilizes a pair of getter/setter functions available as member functions of the traited class. This is a very common concept. The reference type is `std::pair<ValueType (Class::*)() const, void (Class::*)(const ValueType&)>` in this case. Therefore, the constructor accepts a corresponding instance of `std::pair` storing the pointers to the getter and setter functions. The disadvantage of this approach is that it does not allow direct access to the data. On the other hand, it is possible to use this reference to simulate a virtual member, e.g., the case of `velocity` in the example in Code listings 3.10 and 3.1. The definition of the specialization for the pair of functions is in Code listing 3.4.

The last way of member access is also provided by a pair of getter/setter functions. There is one significant difference from the second discussed case. The functions are not member functions of the object. They are global functions which access the member, e.g., `std::get` in the case of the `std::tuple`. This way, it is possible create `MemberAccess` to elements of `std::tuple`. See the example in Code listing 3.5.

3.2 Traits as a Tuple of MemberAccess

In the previous Section 3.1 the concept of the class `MemberAccess` was described. The next step in the construction of class traits is to create the `Traits` class template, which has the following template arguments:

1. `Class`: the type of the traited class or structure,
2. `RefTypes`: variadic argument containing the types of the references to the members of `Class`.

Similarly to the construction of the `MeshElements` class (Section 2.2.4), the structure of `Traits` utilizes a private data structure `MemRefs` automatically generating the container for the stored data. The construction of `MemRefs` utilizes similar processes to the construction of `_MeshElements`. `MemRefs` has two template parameters:

1. `Index`: `unsigned int` index declaring the position and the reference from `RefTypes` to be stored in `MemberAccess`,
2. `Dummy`: an extra type to prevent explicit specialization in the unspecialized enclosing class template `Traits`, which would cause an error [7].

The definition of the `Traits` private member class `MemRefs` is the following. In order to generate the data containing structure, the `MemRefs` class inherits another `MemRefs` with increased `Index` by one until the `Index` reaches the number of `RefTypes` -1 (see lines 14-34 in Code listing 3.6). Each of the contained `MemRefs` has two constant attributes. The first is the `MemberAccess` with type of the reference corresponding to the `Index` named `ref`. The second attribute is `name` and has type of `const char*`.

Let us note that all the attributes of `Traits` and `MemRefs` are declared as `const` due to performance optimizations. This implies the need of initializing all the attributes in the constructor

Code listing 3.3 The specialization of `MemberAccess` for a pointer to member (type `ValueType Class::*`). The class publishes the types of the referenced value `typeValue`, traitled class `Class` and reference `refType`. Next the class have one constant attribute of `refType` named `ref`. This attribute is set in the constructor which accepts the value of reference. Furthermore, the class have the `getValue` and `setValue` member functions. Moreover, as the pointer to member provides possibility to approach the attribute directly, this specialization has member function `getAttr` returning the reference to a member. Finally, all the three member functions have two versions, one for a reference to the instance of `Class` and the other for a pointer to the instance.

```

1  template <typename Class, typename ValueType>
2  class MemberAccess<ValueType Class::*> : public DirectAccess, public ConstGetAccess {
3  public:
4      using typeValue = ValueType;
5      using typeClass = Class;
6      using refType = ValueType Class::*;
7
8  private:
9      // Constant reference possibly resolved at compile time
10     const refType ref;
11 public:
12
13     MemberAccess(refType referenceToMember) : ref(referenceToMember){
14         //ref = referenceToMember;
15     }
16
17     MemberAccess(const MemberAccess<ValueType Class::*>&) = default;
18
19     MemberAccess(MemberAccess<ValueType Class::*&&) = default;
20
21     ValueType getValue(const Class* c) const {
22         return c->*ref;
23     }
24
25     void setValue(Class* c, const ValueType& val) const {
26         c->*ref = val;
27     }
28
29     ValueType& getAttr(Class* c) const {
30         return c->*ref;
31     }
32
33     ValueType getValue(const Class& c) const {
34         return c.*ref;
35     }
36
37     void setValue(Class& c, const ValueType& val) const {
38         c.*ref = val;
39     }
40
41     ValueType& getAttr(Class& c) const {
42         return c.*ref;
43     }
44 };

```

Code listing 3.4 The specialization of `MemberAccess` for a pair of getter/setter member functions. The first function is a getter and the second one is setter of an attribute, e.g., the functions `getVelocity` and `setVelocity` shown in Code listing 3.1. However, the member functions are free to do anything as long as they have the correct arguments. This type of reference does not provide direct access to an attribute, therefore this specialization provides the functions `getValue` and `setValue` only.

```

1  template <typename Class, typename ValueType>
2  class MemberAccess<std::pair<ValueType (Class::*)() const,
3                    void (Class::*)(const ValueType&>>
4      : public ConstGetAccess {
5  public:
6      using typeValue = ValueType;
7      using typeClass = Class;
8
9      using getterType = ValueType (Class::*)() const;
10     using setterType = void (Class::*)(const ValueType&);
11
12 private:
13     getterType const refGet;
14     setterType const refSet;
15
16 public:
17
18     MemberAccess(std::pair<getterType, setterType> getSet)
19         :refGet(getSet.first), refSet(getSet.second){
20         //refGet = getSet.first;
21         //refSet = getSet.second;
22     }
23
24     MemberAccess(const MemberAccess<std::pair<getterType, setterType>>&) = default;
25
26     MemberAccess(MemberAccess<std::pair<getterType, setterType>>&&) = default;
27
28
29     ValueType getValue(const Class* c) const {
30         return (c->*refGet)();
31     }
32
33     void setValue(Class* c, const ValueType& val) const {
34         (c->*refSet)(val);
35     }
36
37     ValueType getValue(const Class& c) const {
38         return (c.*refGet)();
39     }
40
41     void setValue(Class& c, const ValueType& val) const {
42         (c.*refSet)(val);
43     }
44 };

```

Code listing 3.5 An example of making a `MemberAccess` to the first element of a tuple. The specialization of `MemberAccess` for two global functions is applied. The key part is to exactly specify the desired version of the overloaded function. The specification is achieved by typecasting the function pointer. This way, the compiler knows what pointer is requested. Otherwise the compilation fails due to ambiguity.

```

1  // Access to the first element of type double of class std::tuple<double>
2  // Using the static_cast the version of overloaded function get is specified
3  MemberAccess(
4      std::make_pair(static_cast<const double&(*)>(const std::tuple<double>&)>(std::get<0>),
5                    static_cast<double&(*)>(std::tuple<double>&)>(std::get<0>))
6  );

```

Code listing 3.6 The definition of the Traits class template. The Traits class has one attribute of type MemRefs, which is a private member structure. MemRefs builds a system storing the MemberAccess instances for all types in RefTypes structures and the corresponding names. The constructor of MemRefs class enables to declare the attributes as constant. Furthermore, the class Traits provides the basic aliases, i.e., refType (type of single reference), memRefType (corresponding MemberAccess for the given refType), type (the type the refType refers to). Finally, the whole setup is done using the constructor of Traits. Its arguments have to form pairs of names and the corresponding references.

```

1  template<typename Class, typename...RefTypes>
2  class Traits {
3  public:
4      template<unsigned int Index>
5          using refType = typename std::tuple_element<Index, std::tuple<RefTypes...>>::type;
6
7      template<unsigned int Index>
8          using memRefType = MemberAccess<refType<Index>>;
9
10     template <unsigned int Index>
11         using type = typename MemberAccess<refType<Index>>::typeValue;
12
13 private:
14     template<unsigned int Index = 0, typename = void>
15     struct MemRefs: public MemRefs<Index + 1> {
16
17         const MemberAccess<refType<Index>> ref;
18         const char* name;
19
20         template <typename ... REST>
21         MemRefs(const char* n, refType<Index> r, REST... rest)
22             : MemRefs<Index + 1> (rest...), ref(r), name(n){}
23     };
24
25     template<typename Dummy>
26     struct MemRefs<sizeof...(RefTypes) - 1, Dummy>{
27
28         const MemberAccess<refType<sizeof...(RefTypes) - 1>> ref;
29         const char* name;
30
31         MemRefs(const char* n, refType<sizeof...(RefTypes) - 1> r)
32             : ref(r), name(n){}
33     };
34
35     // Declaration of the refs (MemRefs) attribute
36     const MemRefs<0, void> refs;
37 public:
38     // The constructor only initializes the attribute refs
39     template<typename...Refs>
40     Traits(Refs... refsAndNames) : refs(refsAndNames...){}
41
42     // ... **member functions**
43 };

```

of the corresponding class at once. The solution consists in using a variadic constructor accepting all the names and references at once. Then, the first called constructor, i.e., `MemRefs` with `Index` equal to 0, sets the `name` to the first argument and `ref` to the second one. Thanks to the possibility of calling the constructor of the parent class in the constructor of the derived class, the rest of the parameter pack is passed to the constructor of `MemRefs` with `Index` increased by one. In the end, the constructor of the terminating specialization expects exactly two arguments, the first is `name` (`const char*`) and the second is the last type of the `RefTypes` parameter pack. Finally, the data of `Traits` are initialized by passing all parameters given to `Traits` to the `MemRefs` attribute `refs`. See the implementation of the `Traits` in Code listing 3.6.

The `Traits` class provides one member function (`size`) and several member function templates with `unsigned int` parameter named `Index`. The list of the member functions is the following:

- `size`: returns the number of stored references,
- `getReference`: function template returning the `MemberAccess` stored in `refs` at the position given by `Index`,
- `getName`: function template returning the name corresponding to `Index`,
- `getAttr`: function template returning the l-value reference to an attribute of a `Class` instance,
- `getValue`: function template returning the value of an attribute of a `Class` instance,
- `setValue`: function template setting a value to an attribute of a `Class` instance.

The implementation of the `Traits` member functions is in Code listing 3.7.

An example of declaring `Traits` for the class `qties` is presented in Code listing 3.8. However, as is obvious from the example, the definition of the `Traits` instance is very complicated.

3.3 Default Traits

This section describes the way of publishing an instance of creating a globally accessible instance of `Traits` of a traited class. Since these globally defined class traits are utilized by other functions in GTMesh, we call these global class traits *default class traits*. The concept described below allows to have more than one default class traits defined for one single traited class, e.g., class traits utilized for automatic I/O of the traited class. Moreover, for the purpose of the functions working with traited classes, it is necessary to be able to detect whether a class have default class traits defined or not at compile time (see Section 3.3.2). Finally, the definition of the default class traits must be as simple as possible because the definition of an instance of `Traits` is complicated (see Code listing 3.8).

As the instance of `Traits` must be bound to a class, it is not possible to declare the global instance of `Traits` as a global static variable. Moreover, static variables must be initialized in source files, while the GTMesh is implemented in header files only. Therefore, the chosen approach is to create an additional `DefaultTraits` class template. This class template has one template parameter, the type of the traited class. Generally, the `DefaultTraits` class is defined as an empty class, see Code listing 3.9.

The default class traits for a certain class is then created by a specialization of `DefaultTraits` for the corresponding class (see example in Code listing 3.9). This way, the default instance of `Traits` can be obtained by the public static member function `getTraits` of `DefaultTraits<Class>`, where the `Class` is the type of the traited class. In other words, the default traits can be obtained based on the type (information) of the traited class. The usage of the default traits is presented in Code listing 3.10.

Code listing 3.7 The member functions of the `Traits` class. All the member functions except `size` are templates with argument `Index` of a type `unsigned int` specifying the position of requested name or reference. The `size` member function returns the number of stored references. The functions `getName` and `getReference` returns the contents of the `refs` attribute (see Code listing 3.6). The rest of the member functions are designed to simplify access to the members of the instance of the traited class. Note that the method `getAttr` is available only for references with direct access to an attribute.

```

1 // ** Traits member functions **
2 static constexpr unsigned int size(){
3     return sizeof... (RefTypes);
4 }
5
6 template<unsigned int Index>
7 const MemberAccess<refType<Index>> getReference() const {
8     return refs.MemRefs<Index, void>::ref;
9 }
10
11 template<unsigned int Index>
12 type<Index> getValue(Class* c) const {
13     return getReference<Index>().getValue(c);
14 }
15
16 template<unsigned int Index>
17 type<Index> getValue(Class& c) const {
18     return getReference<Index>().getValue(c);
19 }
20
21 template<unsigned int Index>
22 type<Index> getValue(const Class* c) const {
23     static_assert(HasConstGetAccess<memRefType<Index>>::value,
24                 "The current reference to does not provide constant access.");
25     return getReference<Index>().getValue(c);
26 }
27
28 template<unsigned int Index>
29 type<Index> getValue(const Class& c) const {
30     static_assert(HasConstGetAccess<memRefType<Index>>::value,
31                 "The current reference to does not provide constant access.");
32     return getReference<Index>().getValue(c);
33 }
34
35 template<unsigned int Index>
36 void setValue(Class* c, const type<Index>& val) const {
37     getReference<Index>().setValue(c, val);
38 }
39
40 template<unsigned int Index>
41 void setValue(Class& c, const type<Index>& val) const {
42     getReference<Index>().setValue(c, val);
43 }
44
45 template<unsigned int Index>
46 type<Index>& getAttr(Class* c) const {
47     static_assert(IsDirectAccess<memRefType<Index>>::value,
48                 "The current reference to does not provide direct access.");
49     return getReference<Index>().getAttr(c);
50 }
51
52 template<unsigned int Index>
53 type<Index>& getAttr(Class& c) const {
54     static_assert(IsDirectAccess<memRefType<Index>>::value,
55                 "The current reference to does not provide direct access.");
56     return getReference<Index>().getAttr(c);
57 }
58
59 template<unsigned int Index>
60 const char* getName() const {
61     return refs.MemRefs<Index, void>::name;
62 }

```

Code listing 3.8 An example of definition of an instance of `Traits` providing access to `density` and `velocity` of the class `qties` (defined in Code Listing 3.1). The template arguments are set as `decltype` applied to the value of the reference. Notice the complexity of the definition of the instance of `Traits`. Luckily, here there is a pattern in the definition (which is exploited in Section 3.3.3).

```

1 // Declaration of traits for class qties
2 Traits<qties,
3     decltype(&qties::density),
4     decltype(make_pair(&qties::getVelocity, &qties::setVelocity))>
5     attrTraits("density",
6               &qties::density,
7               "velocity",
8               make_pair(&qties::getVelocity, &qties::setVelocity));
9 // Instance of qties class
10 qties qInstance;
11
12 attrTraits.getAttr<0>(qInstance) = 5;
13 attrTraits.setValue<1>(qInstance, Vector<3, double>{0, 1, 2});
14 // The momentum of qInstance is set to {0, 5, 10}
15
16 // Printing the content of qInstance
17 // Prints "density: 5"
18 printf("%s: %f", attrTraits.getName<0>(), attrTraits.getValue<0>(qInstance));
19 // Prints "velocity: {0, 1, 2}"
20 printf("%s: {%f, %f, %f}",
21        attrTraits.getName<1>(),
22        attrTraits.getValue<1>(qInstance)[0],
23        attrTraits.getValue<1>(qInstance)[1],
24        attrTraits.getValue<1>(qInstance)[2]);

```

Code listing 3.9 Presentation of the default class traits concept. The first listing presents the definition of the `DefaultTraits` class template. The class body is empty because it is to be overridden by a specialization for the given traited class the default class traits is created for. The second presents the definition of the default class traits for the `qties` class (see Code Listing 3.1) by specializing `DefaultTraits`. The instance of the global `Traits` is provided by the public static member function `getTraits`.

```

1 // The definition of the DefaultTraits class.
2 // In its specializations, this class publishes the instance of Traits
3 // bound to the Class the specialization is defined for.
4 template<typename Class>
5 class DefaultTraits{};

```

```

1 // Declaration of default class traits
2 template<>
3 class DefaultTraits<qties>{
4     using traitsType Traits<qties,
5         decltype(&qties::density),
6         decltype(make_pair(qties::getVelocity, qties::setVelocity))>;
7
8     static const traitsType getTraits() {
9         return traitsType("density",
10            &qties::density,
11            "velocity",
12            make_pair<&qties::getVelocity, &qties::setVelocity>);
13     }
14     static constexpr unsigned int size() {return traitsType::size();}
15 }

```


Code listing 3.10 An example of definition of default class traits for the class `qties` (see Code Listing 3.1). There are two default class traits defined, the first is the general default class traits and the second is the definition of default input and output class traits. The default class traits is defined using the macro `MAKE_ATTRIBUTE_TRAIT` (see Section 3.3.3). This defines `DefaultTraits` accessing the attributes by member pointers. The second macro `MAKE_CUSTOM_TRAIT_IO` defines a specialization of `DefaultIOTraits` for `qties` class. Thanks to this construction, the class seems to consist of the attributes `density` and `velocity` instead of `density` and `momentum`. Moreover, when the values of `qties` are to be set using `DefaultIOTraits`, it is necessary for the `density` attribute to be set first, otherwise the result in `momentum` will not be as expected. The function `foo` presents the way of data access.

```

1 // Macro creates specialization of DefaultTraits for qties and names
2 // according to the attributes names
3 MAKE_ATTRIBUTE_TRAIT(qties, density, momentum);
4
5 // Macro creates specialization of DefaultIOTraits for the qties class
6 MAKE_CUSTOM_TRAIT_IO(
7     qties,
8     "density", &qties::density,
9     "velocity", std::make_pair(&qties::getVelocity, &qties::setVelocity)
10 );
11
12 void foo () {
13     DefaultTraits<qties>::getTraits().getAttr<0>(qInstance) = 3; //get reference to density
14     DefaultTraits<qties>::getTraits().setValue<1>(qInstance, {3, 6, 9}); // set momentum
15
16     DefaultIOTraits<qties>::getTraits().getValue<1>(qInstance); // {1, 2, 3} get velocity
17     DefaultIOTraits<qties>::getTraits().setValue<1>(qInstance, {1, 1, 1}); // set velocity
18
19     DefaultTraits<qties>::getTraits().getValue<1>(qInstance); // {3, 3, 3} get momentum
20 }

```

There are two basic ways how to make a single public instance of `Traits`. The first is to create a static variable of `Traits` in the specialization, however as already said, the static variables are troublesome. Therefore, we chose to create static member function which returns the default instance of the `Traits` class defined in the body of the function. This function has to be named `getTraits` in every specialization of `DefaultTraits`. An example of an explicit specialization of `Traits` for the class `qties` is in Code listing 3.9. In the end, let us note that the specialization of `DefaultTraits` has to be defined in the global namespace.

3.3.1 DefaultIOTraits and DefaultArithmeticTraits

After constructing the default class traits, it appeared to be useful to have the possibility to create more types of default traits, e.g., one class traits for input and output and another for

Code listing 3.11 Definition of the classes `DefaultIOTraits` and `DefaultArithmeticTraits`. Both the classes inherit the class `DefaultTraits<Class>`. Thanks to this trick, if the `DefaultTraits<Class>` is specialized for the given `Class`, then both the classes by default returns the same instance of `Traits` by `getTraits`. However, these classes are not limited to inherit the `DefaultTraits` class only. If there is an specialization for the `Class` type, it is utilized instead and another instance of `Traits` can be published.

```

1 template<typename Class>
2 class DefaultIOTraits : public DefaultTraits<Class> {};
3
4 template<typename Class>
5 class DefaultArithmeticTraits : public DefaultTraits<Class> {};

```

Code listing 3.12 The implementation of the system checking whether the default class traits are defined for a class (T1). The class `HasDefaultTraits` inherits the class `__has_default_traits` with arguments T1 and `void`. By default, the `__has_default_traits` class inherits `std::false_type`. Then, the check of the member function is done by the specialization which is validly defined if `DefaultTraits<T1>` has the public static member function `getTraits`. If the expression in the `decltype` at line 15, returns valid `typename`, `void_t` maps to `void` and the specialization is valid and applied instead of the default definition. The specialization inherits `std::true_type`, see the lines 10-16. Otherwise, if the tested class does not have default class traits defined, the `DefaultTraits<T1>::getTraits()` causes an error in the construction of the specialization of the `__has_default_traits` class. Hence, the class candidate is ignored and the default definition inheriting `std::false_type` is applied. Thanks to this concept, we are able to detect whether the default `Traits` is defined for a certain class.

```

1 namespace Impl {
2
3 template<typename ...>
4 using void_t = void;
5
6 // General definition is utilized in the case when the specialization is invalid
7 template <typename T1, typename = void>
8 struct __has_default_traits : public std::false_type {};
9
10 // This specialization applies when the function named by getTraits is present in
11 // DefaultTraits<T1> class
12 template <typename T1>
13 struct __has_default_traits <
14     T1,
15     void_t<decltype(DefaultTraits<T1>::getTraits())>
16     > : public std::true_type {};
17 }
18
19
20 template<typename T>
21 struct HasDefaultTraits : public Impl::__has_default_traits<T, void> {};

```

arithmetic operations. Hence, the `GTMesh` library provides two more types of default traits. The first is `DefaultIOTraits` and it is designed to facilitate import and export of the class. This type of default class traits is also utilized by the debugging system described in Section 4.1. The second default traits is `DefaultArithmeticTraits`. Thanks to the optimizations, `Traits` can be used to create generic computational methods for classes with `DefaultArithmeticTraits`, see Section 4.2. This way, one class may have more than one global instance of `Traits` for different purposes.

By default both the `DefaultIOTraits` and `DefaultArithmeticTraits` classes coincide with the default class traits, i.e., `DefaultTraits`. In other words, when the `DefaultTraits<Class>` is specialized, the `DefaultIOTraits<Class>` and `DefaultArithmeticTraits<Class>` have the same definition. This is achieved by inheritance of `DefaultTraits<Class>` in the definition of the both classes, see Code listing 3.11. Moreover, this approach still allows to override the definition with custom specialization (see Section 4.2). The specialization is done in the same way as presented in Code listing 3.9.

In the terms of `GTMesh` the *default class traits* is synonym for certain specialization of the three classes `DefaultTraits`, `DefaultIOTraits`, `DefaultArithmeticTraits` (see Table 3.1).

3.3.2 Existence of Default Class Traits

The next problem is to detect whether the default class traits are specialized for a certain class, i.e., create a type trait which accepts a class (`typename`) and returns `true` if the default class traits has a specialization declared for the particular class, or `false` otherwise. This

Code listing 3.13 Three possible ways of default Traits definition by utilizing the helper macros. All of the macros define the same default class traits for the class `flowQ`. The differences between the individual applications of the macros are described in Section 3.3.3. The second listing presents the application of macros if the traited class is a template.

```

1 struct flowQ{
2     double rho; // density
3     Vector<3, double> p; // momentum
4 };
5
6 MAKE_CUSTOM_TRAIT(flowQ, "density", &flowQ::rho, "momentum", &flowQ::p);
7 MAKE_NAMED_ATTRIBUTE_TRAIT(flowQ, "density", rho, "momentum", p);
8 MAKE_ATTRIBUTE_TRAIT(flowQ, rho, p);

1 template<unsigned int Dimension, typename Real>
2 struct flowQ{
3     Real rho; // density
4     Vector<Dimension, Real> p; // momentum
5 };
6
7 MAKE_CUSTOM_TEMPLATE_TRAIT(
8     (flowQ<Dimension, Real>),
9     (unsigned int Dimension, typename Real),
10    "density", (&flowQ<Dimension, Real>::rho), "momentum", (&flowQ<Dimension, Real>::p)
11 );
12
13 MAKE_ATTRIBUTE_TEMPLATE_TRAIT(
14     (flowQ<Dimension, Real>),
15     (unsigned int Dimension, typename Real),
16     rho, p
17 );

```

is achieved by the SFINAE paradigm utilized in the class `HasDefaultTraits` [5]. This class simply checks whether `DefaultTraits` of the given class has the member function `getTraits` defined, which is mandatory for the specializations of default class traits. Here the empty definition of the `DefaultTraits` class comes in handy. If the default class traits is not defined, the `DefaultTraits<Class>` is empty, thus it does not have the member function `getTraits`. The implementation of the `HasDefaultTraits` class is described in Code listing 3.12.

Finally, there are classes `HasDefaultIOTraits` and `HasDefaultArithmeticTraits` detecting whether the given class has the corresponding default class traits defined. These classes have an implementation similar to `HasDefaultTraits`.

3.3.3 Macros Creating Default Class Traits

The last requested functionality of the class traits concept is a single line definition of the default class traits. According to Code listing 3.9, the definition of the default class traits requires a relatively complex construction and definition of some mandatory members of the specialization. However, thanks to the pattern in the definition of the specialization, a variadic macro that expands into the correct specialization can be developed. GTMesh provides three macros to simplify the definition of the default class traits.

- The first and the most generic macro is named `MAKE_CUSTOM_TRAIT`. This macro accepts the name of the class as the first argument, and the remaining parameters are pairs of names and values of references.
- The second macro `MAKE_NAMED_ATTRIBUTE_TRAIT` is more specialized. It creates member references only, but allows to define custom names for them. On the other hand, it does not require full references to the member attributes, only attributes names are expected.

Code listing 3.14 An example of creating the system exporting any expression using the macro `FOR_EACH`. In this case, the macro `PRINT` expands in the call of the function `Print` given the pairs of stringized arguments and the arguments themselves. The output of the call of the `PRINT` at line 20 is shown below. The advantage of this approach is that the expression is automatically exported with the result. This concept is extended in Section 4.1.

```

1 // prints all the names and variables
2 template<typename T, typename... Rest>
3 void Print(const char* varName, const T& var, const Rest& ... rest){
4     Print(varName, var);
5     Print(rest...);
6 }
7
8 template<typename T>
9 void Print(const char* varName, const T& var){
10     std::cout << varName << ":␣" << var << '\n';
11 }
12
13 #define STRVAR(expr) #expr, expr
14
15 #define PRINT(...) Print(FOR_EACH(STRVAR, __VA_ARGS__))
16
17 void foo() {
18     double a = 1, b = 2.5, c = 6.8;
19
20     PRINT(a,b, 10 * c);
21     // Primarily expands in Print(STRVAR(a), STRVAR(b), STRVAR(10 * c))
22     // Then expands in Print("a", a, "b", b, "10 * c", 10 * c)
23 }

```

```

1 a: 1
2 b: 2.5
3 10 * c: 68

```

- The last macro `MAKE_ATTRIBUTE_TRAIT` expects only the name of the class to be traited and a list of the attributes. The names are set to the names of the attributes.

The presented macros expand into specializations of `DefaultTraits`. In order to specialize `DefaultIOTraits` or `DefaultArithmeticTraits`, one shall use `MAKE_CUSTOM_TRAIT_IO` or `MAKE_CUSTOM_TRAIT_ARITHMETIC` respectively. The rest of the types of macros are named in the same way. Further in this work, we will call this group of macros `MAKE_TRAIT`.

According to limitations of macro expansion in processing variadic arguments, the limit on the number of arguments of the macros mentioned above is set to 20 names and 20 attributes (40 arguments in total). Those limitations will be obvious from the description of the implementation. An example of the definition of the default `Traits` using the presented macros is in Code listing 3.13.

As shown in Code listing 3.13, `GTMesh` provides macros for definition of specialization of `Traits` for class templates. The macros names have “`TEMPLATE_`” added before “`TRAIT`”, e.g. `MAKE_CUSTOM_TEMPLATE_TRAIT`. These macros expect the name of the class with the template parameters as the first parameter and the list of the template parameters itself as the second parameter. It is important to enclose the template arguments into round brackets to be passed as one argument of the macro. For better orientation in the notation, see Table 3.1.

Variadic Macro `FOR_EACH` In order to create the macros generating the default class traits specializations, the macro `FOR_EACH` is to be implemented first. This macro expands any other given macro accepting one argument for all remaining arguments given to the `FOR_EACH` macro, i.e., `FOR_EACH(MACRO, x1, x2, ..., xN)` expands in `MACRO(x1)`, `MACRO(x2)`, ..., `MACRO(xN)`. The effect of the `FOR_EACH` macro can be demonstrated on creating automatic export shown in Code listing 3.14.

Code listing 3.15 All macros utilized in the construction of macro `FOR_EACH`. The first four macros together are responsible for counting of the passed arguments (see the example in Code listing 3.16). The following macro `CONCATENATE` concatenates the expressions passed in `arg1` and `arg2` together. Then, the next four macros starting with `FOR_EACH_00` realize the application of the expression `what` on the given arguments separated by commas (see example in Code listing 3.17). The last two macros `FOR_EACH` and `FOR_EACH_` together count the number of arguments and then concatenate the number of arguments with expression `FOR_EACH_`, see the example in Code listing 3.18. This listing is limited to 3 arguments for demonstration purpose. In GTMesh, the `FOR_EACH` macro is able to manage up to 40 arguments. The purpose of the `FOR_EACH_NARG_` is to ensure the expansion of the `FOR_EACH_RSEQ_N` macro. Without `FOR_EACH_NARG_`, the `FOR_EACH_RSEQ_N` macro would be passed as a single argument into `FOR_EACH_ARG_N` which results in an compilation error.

```

1 #define FOR_EACH_RSEQ_N 03, 02, 01, 00
2 #define FOR_EACH_ARG_N(_1, _2, _3, N, ...) N
3 #define FOR_EACH_NARG(...) FOR_EACH_ARG_N(__VA_ARGS__)
4 #define FOR_EACH_NARG(...) FOR_EACH_NARG(__VA_ARGS__, FOR_EACH_RSEQ_N)
5
6 #define CONCATENATE(arg1, arg2) arg1##arg2
7
8 #define FOR_EACH_00(what, ...)
9 #define FOR_EACH_01(what, x, ...) what(x)
10 #define FOR_EACH_02(what, x, ...) what(x), FOR_EACH_01(what, __VA_ARGS__)
11 #define FOR_EACH_03(what, x, ...) what(x), FOR_EACH_02(what, __VA_ARGS__)
12
13 #define FOR_EACH_(N, what, ...) CONCATENATE(FOR_EACH_, N)(what, __VA_ARGS__)
14 #define FOR_EACH(what, ...) FOR_EACH_(FOR_EACH_NARG(__VA_ARGS__), what, __VA_ARGS__)

```

Code listing 3.16 Example of expansion of the macro `FOR_EACH_NARG` counting the number of arguments. The macro `FOR_EACH_NARG` eventually expands in `FOR_EACH_ARG_N` with the given arguments and the appended reverse sequence defined in `FOR_EACH_RSEQ_N`. According to the definition of `FOR_EACH_ARG_N` recalled at line 4, the argument 02 is passed as argument N and is returned. The definitions of the macros applied here are in Code listing 3.15.

```

1 FOR_EACH_NARG(arg1, arg2)
2 -> FOR_EACH_NARG(arg1, arg2, FOR_EACH_RSEQ_N)
3 -> FOR_EACH_ARG_N(arg1, arg2, 03, 02, 01, 00)
4 // FOR_EACH_ARG_N(_1 , _2 , _3, N, ...) N
5 -> 02 is returned from the macro

```

From the implementation point of view, the preprocessor does not allow recursion in macro expansion. Luckily, inside a macro another macro can be expanded. Then, the final solution consists in counting of the number of the passed arguments and then passing the arguments to a macro corresponding to the number of arguments. The whole construction is discussed in detail below. Before we start with the description, let us mention that the variable arguments in macros are substituted for the keyword `__VA_ARGS__` written in the definition of a variadic macro.

The first step is to prepare a system counting the number of arguments of the macro. This is achieved by cooperation of several macros. The first macro is named by `FOR_EACH_RSEQ_N` and it defines a decreasing sequence of numbers (see line 1 in Code listing 3.15). The next macro `FOR_EACH_ARG_N` defines a system counting the number of arguments, see the definition at line 2 in Code listing 3.15. This macro accepts a sequence of arguments with length up to the number of arguments starting with `_`, e.g., `_00` and expects the prepared sequence to be appended to the arguments. This macro then publishes the Nth argument corresponding to the number arguments thanks to the appended sequence. Next, the remaining two macros `FOR_EACH_NARG_` and `FOR_EACH_NARG` ensure the correct expansion and application of the macro `FOR_EACH_ARG_N`.

Code listing 3.17 A demonstration of expansion of macro `FOR_EACH_03`.

```
1 FOR_EACH_03(STRVAR, a, b, 10 * c)
2 -> STRVAR(a), FOR_EACH_02(STRVAR, b, 10 * c)
3 -> STRVAR(a), STRVAR(b), FOR_EACH_01(STRVAR, 10 * c)
4 -> STRVAR(a), STRVAR(b), STRVAR(10*c)
```

Code listing 3.18 An example of expansion of macro `FOR_EACH` applied to expressions `STRVAR`, `a`, `b` and `10*c`. The expansion of `FOR_EACH_03(STRVAR, a, b, 10 * c)` is presented in Code listing 3.17.

```
1 FOR_EACH(STRVAR, a, b, 10 * c)
2 // FOR_EACH_(FOR_EACH_NARG(a, b, 10 * c), STRVAR, a, b, 10 * c)
3 -> FOR_EACH_(03, STRVAR, a, b, 10 * c)
4 // CONCATENATE(FOR_EACH_, 03)(STRVAR, a, b, 10 * c)
5 -> FOR_EACH_03(STRVAR, a, b, 10 * c)
6 // ...
```

These macros together accept the arguments and pass them to `FOR_EACH_ARG_N` with the sequence appended, see the lines 3 and 4 in Code listing 3.15. Generally, the macro `FOR_EACH_NARG` expands to the element of the sequence `FOR_EACH_RSEQ_N` corresponding to the number of the given arguments, see the example of application of `FOR_EACH_NARG` in Code listing 3.16. This system is able to count the number of passed arguments up to the number of the prepared arguments in `FOR_EACH_ARG_N` and `FOR_EACH_RSEQ_N`. In the example of the definition shown Code listing 3.15, the maximum number of arguments is three, however in the GTMesh library, this system of macros is able to manage up to 40 arguments.

Next, as the system counting the number of the arguments is complete, we define the macro `CONCATENATE` (see the definition at line 6 in Code listing 3.15). This macro concatenates the expressions passed as `arg1` and `arg2` omitting any blank spaces between them, e.g., `CONCATENATE(get, Value)` will expand in `getValue`. This macro will be used to concatenate the keyword `FOR_EACH_` with the number of arguments, e.g., `FOR_EACH_02`. This way, the name of newly expanded macro will be created.

The next step is to prepare a system of macros `FOR_EACH_#N`, where the `#N` matches the elements of the `FOR_EACH_RSEQ_N` sequence. See the definition at lines 8-11 in Code listing 3.15. These macros have the following arguments:

1. **what**: the expression to be applied on argument `x`,
2. **x**: the currently processed argument,
3. **variadic arguments**: list of arguments to be passed further to `FOR_EACH_#N-1`.

The **what** argument is to be the name of a function or a function-like macro that accepts one parameter. Each macro `FOR_EACH_#N` except `FOR_EACH_01` and `FOR_EACH_00` then expands in `what(x)`, `FOR_EACH_#N-1(what, __VA_ARGS__)`, i.e., the next macro is given the **what** expression and the rest of the parameters separated by commas. The `FOR_EACH_01` expands in `what(x)` only, this way the recursive expansion is stopped. The definition of the macro `FOR_EACH_00` is left blank. The purpose of this macro is to prevent an error when the `FOR_EACH` is given no arguments. For a better understanding of the functionality of the `FOR_EACH` macro, see Code listing 3.17.

Now, all the components constructing the `FOR_EACH` macro are prepared. The last two macros `FOR_EACH_` and `FOR_EACH` are to be defined, see the definition in Code listing 3.15. These two macros ensure the correct usage of the previously discussed macros. The first mentioned macro `FOR_EACH_` accepts the number of arguments counted by `FOR_EACH_NARG` and concatenates it with word `FOR_EACH_`. Then it passes the **what** argument followed by the pack of arguments to the `FOR_EACH_#N` macro. Finally, the macro `FOR_EACH` counts the number of passed arguments

Code listing 3.19 The definition of the macro `FOR_EACH_3ARGS_1STAT`. This macro applies the macro `what` to the given argument `x_stat` and each two arguments taken from the remaining argument sequence `x1` and `x2`.

```

1 #define FOR_EACH_3ARGS_1STAT_00(what, ...)
2 #define FOR_EACH_3ARGS_1STAT_02(what,x_stat, x1, x2, ...)  what(x_stat, x1, x2)
3 #define FOR_EACH_3ARGS_1STAT_04(what,x_stat, x1, x2, ...)  what(x_stat, x1, x2), \
4   FOR_EACH_3ARGS_1STAT_02(what, x_stat, __VA_ARGS__)
5
6 #define FOR_EACH_3ARGS_1STAT_(N, what, x_stat, ...) \
7   CONCATENATE(FOR_EACH_3ARGS_1STAT_, N)(what, x_stat, __VA_ARGS__)
8
9 #define FOR_EACH_3ARGS_1STAT(what, x_stat, ...) \
10  FOR_EACH_3ARGS_1STAT_(FOR_EACH_NARG(__VA_ARGS__), what, x_stat, __VA_ARGS__)

```

Code listing 3.20 An example of application of macros `FOR_EACH_2ARGS` and `FOR_EACH_3ARGS_1STAT`. This example presents automatic processing of the arguments which are typically passed to the macros defining default class traits. The first example creates a sequence of types deduced from the value of references. The second example presents the transformation of names of attributes to pointers to members.

```

1 // Omit the odd arguments
2 #define IMPL_MEMREF_TYPE_CUSTOM(name, memberRef) decltype(memberRef)
3
4 FOR_EACH_2ARGS(IMPL_MEMREF_TYPE_CUSTOM, name1, ref1, name2, ref2)
5 -> decltype(ref1), decltype(ref2)
6
7 // Prepend the static argument to even arguments in __VA_ARGS__
8 #define IMPL_NAME_AND_REF(Class, name, member) name, &Class::member
9
10 FOR_EACH_3ARGS_1STAT(IMPL_NAME_AND_REF, ClassName, name1, member1, name2, member2)
11 -> name1, &ClassName::member1, name2, &ClassName::member2

```

and passes the information further into the `FOR_EACH_` macro. For better understanding, see the example explaining the expansion of the `FOR_EACH` macro in Code listing 3.18.

Moreover, for the needs of the generation of default Traits two more types of the `FOR_EACH` macro were developed. The first is `FOR_EACH_2ARGS`. As the name suggests, this macro applies `what` to two arguments in row. The last developed macro is `FOR_EACH_3ARGS_1STAT`. This macro expands in `what(x_stat, x1, x2)`, where the `x_stat` is the same in all expansions of `what`, the `x1, x2` are the first two expressions from the variable argument (see the definition in Code listing 3.19). See the example explaining the possible usage of the discussed macros in Code listing 3.20.

MAKE_TRAIT Macros Definition Now, thanks the macro `FOR_EACH` we are able to create the `MAKE_TRAIT` macros (see Table 3.1). The macros `MAKE_TRAIT` generate the specialization of default class traits based on the given class, names and references. In this section the exact implementation of `MAKE_TRAIT` macros is presented.

At first, the macros applied in the `FOR_EACH` macros are to be defined.

- `IMPL_MEMREF_TYPE_CUSTOM`: accepts 2 arguments, `name` and `memberRef`. Then, it expands to `decltype(memberRef)` (see the example in Code listing 3.20). This macro is to be used to specify the template arguments of the global Traits in default Traits.
- `IMPL_NAME_AND_REF`: has 3 arguments, `Class`, `name` and `member`. This macro expands to `name, &Class::member`. This macro is designed to be used in `FOR_EACH_3ARGS_1STAT` (see the example in Code listing 3.20).

Code listing 3.21 The definition of the macros necessary for the generation of the default class traits. The macro `IMPL_MAKE_CUSTOM_TRAIT` accepts the name of the traits to be specialized (`TraitName`), the traitled class (`Class`) and a sequence of names and references. This macro then creates the appropriate explicit specialization of class passed through `TraitName` (i.e., `DefaultTraits`, `DefaultIOTraits`, `DefaultArithmeticTraits`) for the traitled class.

```

1 #define IMPL_MEMREF_TYPE_CUSTOM(name, memberRef) decltype(memberRef)
2 #define IMPL_NAME_AND_REF(Class, name, member) name, (&Class::member)
3 #define IMPL_NAME_ATT(attribute) #attribute, attribute
4
5 #define IMPL_MAKE_CUSTOM_TRAIT(TraitName, Class, ...) \
6 template<> \
7 class TraitName<Class>{ \
8 public: \
9     using traitsType = Traits<Class, FOR_EACH_2ARGS(IMPL_MEMREF_TYPE_CUSTOM, __VA_ARGS__)>; \
10    static const traitsType getTraits() {return traitsType(__VA_ARGS__);} \
11    static constexpr unsigned int size() {return traitsType::size();} \
12 }

```

Code listing 3.22 The definition of the macros creating specialization of the class `Traits` for a traitled class. The macro `MAKE_CUSTOM_TRAIT` simply utilizes the macro `IMPL_MAKE_CUSTOM_TRAIT` with `Traits` passed as `TraitName`. The macro `MAKE_NAMED_ATTRIBUTE_TRAIT` transforms the names of the attributes into member pointers using `IMPL_NAME_AND_REF` defined in Code listing 3.21 and passes the processed arguments to `MAKE_CUSTOM_TRAIT`. Finally, the `MAKE_ATTRIBUTE_TRAIT` creates strings from the names of the given attributes and passes the arguments to `MAKE_NAMED_ATTRIBUTE_TRAIT`. The same construction is utilized in the definition of macros creating specializations of `DefaultIOTraits` and `DefaultArithmeticTraits`.

```

1 #define MAKE_CUSTOM_TRAIT(Class, ...) \ // defining specialization for DefaultTraits
2     IMPL_MAKE_CUSTOM_TRAIT(DefaultTraits, Class, __VA_ARGS__)
3
4 #define MAKE_NAMED_ATTRIBUTE_TRAIT(Class, ...) \
5     MAKE_CUSTOM_TRAIT(Class, FOR_EACH_3ARGS_1STAT(IMPL_NAME_AND_REF, Class, __VA_ARGS__))
6
7 #define MAKE_ATTRIBUTE_TRAIT(Class, ...) \
8     MAKE_NAMED_ATTRIBUTE_TRAIT(Class, FOR_EACH(IMPL_NAME_ATT, __VA_ARGS__))

```

- `IMPL_NAME_ATT`: expands into a string literal of the given expression and the expression itself, separated by a comma.

The definition of the three macros are at lines 1-3 in Code listing 3.21.

While more types of the default traits exist in `GTMesh`, the definition of the specializations is the same for all of them. This macro accepts the name of the class traits to be specialized (e.g., `DefaultTraits` or `DefaultIOTraits`) and the name of the traitled class followed by names and references. A generic macro `IMPL_MAKE_CUSTOM_TRAIT` expands in the specialization of the class with the defined member alias `traitsType` and two member functions `getTraits` and `size`. The `traitsType` specifies the type of the globally defined `Traits`, e.g., `Traits<flowQ, double flowQ::*, Vector<3, double> flowQ::*>`. Next, `getTraits` returns an instance of the global `Traits` with the given names and references. Finally, the member function `size` returns the number of members. The definition of the macro `IMPL_MAKE_CUSTOM_TRAIT` is presented in Code listing 3.21.

The macros creating default class traits are defined as shown in Code listing 3.22. For better understanding to the expansion of the specialized macros accepting only the names of the attributes, see the example in Code listing 3.23. In the end, let us note that the macros defining the specialization of `DefaultIOTraits` and `DefaultArithmeticTraits` follow the same naming convention. The macro names are appended by “`_IO`” or “`_ARITHMETIC`”, respectively,

Code listing 3.23 Demonstration of the expansion of MAKE_ATTRIBUTE_TRAIT which is given the class flowQ and attributes names rho and p.

```

1 MAKE_ATTRIBUTE_TRAIT(flowQ, rho, p);
2 // MAKE_NAMED_ATTRIBUTE_TRAIT(flowQ, FOR_EACH(IMPL_NAME_ATT, rho, p))
3 -> MAKE_NAMED_ATTRIBUTE_TRAIT(flowQ, "rho", rho, "p", p);
4 // MAKE_CUSTOM_TRAIT(flowQ, FOR_EACH_3ARGS_1STAT(IMPL_NAME_AND_REF, flowQ, "rho", rho, "p", p))
5 -> MAKE_CUSTOM_TRAIT(flowQ, "rho", &flowQ::rho, "p", &flowQ::p);
6 -> IMPL_MAKE_CUSTOM_TRAIT(DefaultTraits, flowQ, "rho", &flowQ::rho, "p", &flowQ::p);
7 // Expands in
8 template<>
9 class DefaultTraits<flowQ>{
10 public:
11     using traitsType = Traits<flowQ, decltype(&flowQ::rho), decltype(&flowQ::p)>;
12     static const traitsType getTraits() {
13         return traitsType("rho", &flowQ::rho, "p", &flowQ::p)
14     }
15     static constexpr unsigned int size() {return traitsType::size();}
16 };

```

Code listing 3.24 The upper listing presents the definition of the UNWRAP macro. This macro extract the content enclosed in round brackets. This is achieved by appending the argument right after the PASS macro. If the argument itself is enclosed in brackets, the preprocessor continues with the expansion of the macro PASS. The macro PASS expands its content regardless of the number of the arguments passed. When the argument of the UNWRAP macro is not enclosed in brackets, the PASS arg does not result in function-like macro invocation and compilation fails. The lower listing presents the application of the macro UNWRAP.

```

1 #define PASS(...) __VA_ARGS__
2 #define UNWRAP(arg) PASS arg

```

```

1 UNWRAP((Class<T1, T2, T3>))
2 -> PASS(Class<T1, T2, T3>)
3 -> Class<T1, T2, T3>

```

e.g., MAKE_CUSTOM_TRAIT_IO.

MAKE_TEMPLATE_TRAIT Macros Definition The term MAKE_TEMPLATE_TRAIT denotes a group of macros designed to create default class traits for class templates, as presented in Code listing 3.13 (see Table 3.1). The implementation of the MAKE_TEMPLATE_TRAIT macros is very similar to the implementation of MAKE_TRAIT. However, there is a caveat connected with passing complicated arguments as the whole name of class template or the template parameters declaration. The preprocessor interprets a comma as an argument delimiter. For example, the string flowQ<Dimension, Real> is split into 2 arguments flowQ<Dimension and Real>. This problem can be solved by wrapping the argument into round brackets. Then the whole expression is understood as one parameter.

The fact that we passed any parameter in brackets comes with another problem. C++ does not allow the syntax &(Class)::member, where the presence of the brackets causes a compilation error. Thus, it is necessary to develop a construction able to unwrap the content in the brackets. This is done by a macro named UNWRAP presented in Code listing 3.24.

Finally, with minor changes compared to IMPL_MAKE_CUSTOM_TRAIT, it is possible to define IMPL_MAKE_CUSTOM_TEMPLATE_TRAIT. The implementation is shown in Code listing 3.25.

Code listing 3.25 The definition of the `MAKE_TEMPLATE_TRAIT` macros. The definitions are similar to the `MAKE_TRAIT` macros (see Code listings 3.21, 3.22). These macros additionally accept a sequence of template arguments and the template declaration of the traitled class. The `UNWRAP` macro is utilized to extract the class name and template parameters from the bracket enclose. Thus, both the class name and the template parameters must be passed enclosed in brackets even if there is only one template argument and the expressions do not contain commas; see Code listing 3.13.

```

1  #define IMPL_NAME_AND_REF_TEMPLATE(Class, name, member) name, (&UNWRAP(Class)::member)
2
3  #define IMPL_MAKE_CUSTOM_TEMPLATE_TRAIT(TraitName, TemplateParameters, Class, ...) \
4  template<UNWRAP(TemplateParameters)> \
5  class TraitName<UNWRAP(Class)>{ \
6  public: \
7      using traitsType = ::Traits< UNWRAP(Class), \
8          FOR_EACH_2ARGS(IMPL_MEMREF_TYPE_CUSTOM, __VA_ARGS__) >; \
9      static const traitsType getTraits() {return traitsType(__VA_ARGS__);} \
10     static constexpr unsigned int size() {return traitsType::size();}\
11 }
12
13 #define MAKE_CUSTOM_TEMPLATE_TRAIT(Class, TemplateParameters, ...) \
14 IMPL_MAKE_CUSTOM_TEMPLATE_TRAIT(DefaultTraits, TemplateParameters, Class, __VA_ARGS__)
15
16 #define MAKE_NAMED_ATTRIBUTE_TEMPLATE_TRAIT(Class, TemplateParameters, ...) \
17     MAKE_CUSTOM_TEMPLATE_TRAIT( Class, \
18         TemplateParameters, \
19         FOR_EACH_3ARGS_1STAT( IMPL_NAME_AND_REF_TEMPLATE, \
20             Class, \
21                 __VA_ARGS__ ) )
22
23 #define MAKE_ATTRIBUTE_TEMPLATE_TRAIT(Class, TemplateParameters, ...) \
24     MAKE_NAMED_ATTRIBUTE_TEMPLATE_TRAIT( Class, \
25         TemplateParameters, \
26         FOR_EACH(IMPL_NAME_ATT, __VA_ARGS__)

```

Chapter 4

Class Traits Applications

In the previous Chapter 3, the concept of class traits was presented. The aim of this chapter is to present the possibilities provided by the developed class traits. As was already mentioned at the beginning of Chapter 3, one of the motivations was to enable writing generic code for input and output of data structures. In this manner, a debugging (logging) system has been developed. Let us note that this debugging system was very helpful while developing GTMesh.

The debugging system was utilized in the development of every tool of GTMesh both for logging the processes in the called functions and checking the expected results (calculated in advance for the simple test cases). When the tool was working properly, the logging from the functions was deleted. However, the tests were preserved in order to check the correct results in the context of future changes. Finally, as the project was hosted at GitLab [12], we utilized the approach of the continuous integration (CI) [10]. The prepared tests were transformed into test suites and test cases utilizing the GTest library [11].

Once the class traits were optimized to have the same performance as explicitly written code, it was possible to create automatically generated arithmetic operations for the traited classes, as explained in Section 4.2.

4.1 Debugging System and Automatic Data I/O

The first application of the class traits is the automatic logging system. This system is designed to export any variable based on the type or interface, i.e., exploiting the fact that the object can be iterated, has a subscript operator etc. Here the use of `DefaultIOTraits` is only small part of the concept, however it enables the debugging system to export any traited class in JSON (javascript object notation) format.

The aim of the logging tool is to be able to log variables of almost any type, e.g., `std::vector`, `std::map`, traited class, or a combination there of.

4.1.1 The VariableExport Class

The class `VariableExport` is responsible for an export of almost any type of argument. This is achieved by the only public static member function `exportVariable` with many overloads. The generality of `exportVariable` consists in processing the variable based on its interface. `exportVariable` distinguishes 6 basic categories of variables:

1. *traited*: the type has the corresponding `DefaultIOTraits` specialization defined (see Section 3.3.1),
2. *string*: the type is presented as text, i.e., `std::string`, `const char*`, `char`,
3. *exportable*: there is an overloaded `operator<<` for `std::ostream` and the type of the given variable,

Code listing 4.1 The demonstration of usage of the debugging tool in GTMesh. The first listing presents an example `cpp` file, whereas the second one presents the output of the code. At first, the program prepares the variables for the export, i.e., `list` and `map`. Then, using the macro `DBGVAR`, the variables `list` and `map` together with an instance of the structure `Data` are logged. The export at line 24 can be recognized by the term `<<< 24 >>>` in the output. Then, the program reports the execution of line 25 by the macro `CHECKLINE`. Next, the `DBGMSG` writing a message is followed by next `CHECKLINE`. The execution ends at line 28, where the incorrect access to an element of the a vector throws an exception. The `DBGTRY` macro catches the exception and connects the information about the place where the exception occurred with the description of the exception.

```

1  #include "[[path]]/src/Debug/Debug.h"
2  #include <vector>
3  #include <map>
4  #include <string>
5
6  using namespace std;
7
8  struct InnerData {
9      std::vector<char> s = {'h','e','l','l','o'};
10 };
11 MAKE_ATTRIBUTE_TRAIT(InnerData, s);
12
13 struct Data {
14     int data = 42;
15     InnerData d;
16 };
17 MAKE_ATTRIBUTE_TRAIT(Data, data, d);
18
19
20 int main(int argc, char* argv[])
21 {
22     std::vector<std::string> list = {"This is", "absolutely awesome", "debugger!"};
23     std::map<std::string, std::vector<int>> map = {"odd", {1,3,5}}, {"even", {2,4,6}};
24     DBGVAR(list, map, Data());
25     DBGCHECK;
26     DBGMSG("Almost at the end of the program");
27     DBGCHECK;
28     DBGTRY(std::vector<int>({1,2,3}).at(4))
29 }

```

```

1  == main.cpp << 24 >> [[ list ]] ==> [ "This is", "absolutely awesome", "debugger!" ]
2  == main.cpp << 24 >> [[ map ]] ==> [ { "even": [ 2, 4, 6 ]}, { "odd": [ 1, 3, 5 ]} ]
3  == main.cpp << 24 >> [[ Data() ]] ==> { "data" : 42, "d" : { "s" : [ "h", "e", "l", "l", "o" ] } }
4
5  -- main.cpp << 25 >> ==> "check line" <==
6  ++ main.cpp << 26 >> ==> "Almost at the end of the program" <==
7  -- main.cpp << 27 >> ==> "check line" <==
8  !! main.cpp << 28 >> ==> "something went wrong in try block: vector::_M_range_check: __n (
9  which is 4) >= this->size() (which is 3)" <==

```

Code listing 4.2 The definition of a custom type trait `IsIterable` declaring whether a type has an iterator provided. The class `IsIterable` utilizes the implementation in the `__is_iterable` class. The check is performed by determining whether the given type has the `begin` and `end` member functions. The presence of the member function is done by testing of the result type of the function called on an instance obtained by `std::declval`. If the given type `T1` has both the `begin` and `end` member functions, the generated specialization (`__is_iterable<T1, void>`) is valid and utilized. In the opposite case, the specialization is removed from the candidate list. This way, `IsIterable` inherits `std::true_type` if the type `T1` satisfies the conditions and `std::false_type` otherwise.

```

1 namespace Impl {
2 template <typename T1, typename = void>
3 struct __is_iterable : public std::false_type {};
4
5 template <typename T1>
6 struct __is_iterable< T1,
7     void_t< decltype(std::declval<const T1&>().begin()),
8         decltype (std::declval<const T1&>().end()) >
9     > : public std::true_type {};
10 }
11
12 template <typename T1>
13 struct IsIterable : public Impl::__is_iterable<T1>
14 {};

```

4. *pair*: the variable is of the type `std::pair`,
5. *iterable*: the object has `begin` and `end` member functions,
6. *indexable*: the object has a subscript operator and member function `size` defined.

In case the variable falls into more than one category, the categories are given priorities. The priorities correspond to the order as the categories were defined above. The reason to setup the priority list is to prevent an ambiguity when calling the `exportVariable` function. The overloads of `exportVariable` utilize a combination of different argument types and the SFINAE. The `exportVariable` functions have a default (capture) implementation, which is engaged in the case that the interface of the passed variable is not recognized. Therefore, the build of `exportVariable` never fails, but instead of the content of the variable, the `exportVariable` writes that the passed argument can not be exported. See the implementation of several overloads of `exportVariable` in Code listing 4.3.

As an example of the additional type traits utilized in overloading `exportVariable`, we present the detection of the iterator interface. The type trait is called `IsIterable` and its implementation is very similar to the implementation of `HasDefaultTraits`, see Code listing 4.2. The type is tested for the presence of the `begin` and `end` member functions. The other implemented type traits, i.e., `IsExportable` or `IsIndexable`, are constructed analogically.

The traitled classes are detected using `HasDefaultIOTraits`. The traitled class is then exported as a comma-separated list of names and values separated by a colon and enclosed in braces. For an example see the lines 3 and 4 in output in Code listing 4.1. This result is achieved by utilizing the class `PrintClass`. The complete system exporting traitled classes is shown in Code listing 4.4.

4.1.2 Loggers

The debugging tool in GTMesh offers several classes providing different output of the debug log. Each logger class cooperates with a variadic macro that simplifies the usage of the tool. Moreover, it automatically embeds the expression, line and file to the log which improves the readability of the log.

Code listing 4.3 The definitions of several overloads of the `exportVariable` function. The first definition accepts any arguments and has the lowest priority in the candidate list. This function writes into the given output stream that the given argument is not exportable. The second definition is applied if the given argument can be passed to the stream directly using `operator<<`. In the second definition, there are several exceptions in order to handle some special cases. In the case of argument of type `bool`, `true` or `false` is written instead of `1` or `0`. The case of a traited class is discussed in separate Code listing 4.4. The last exception are text types; these variables are exported in quotes. The last presented definition of `exportVariable` applies to an iterable variable. The values of the iterable variable are exported as comma separated list enclosed in square brackets. Notice that the overloading is based on the SFINAE paradigm where the return type is valid if the argument type has the requested qualities.

```

1 // The VariableExport class with static exportVariable member function
2 template <VARIABLE_EXPORT_METHOD target = VARIABLE_EXPORT_METHOD::ostream>
3 struct VariableExport {
4
5 // The capture for variables with unrecognized types
6 static void exportVariable(std::ostream& ost, ...)
7 {
8     ost << "\"variable is not exportable\"" << std::endl;
9 }
10
11 // The overoad for T exportable
12 template<typename T>
13 static auto exportVariable(std::ostream& ost, const T& b)
14     -> typename std::enable_if<
15     IsExportable<T>::value &&
16     !std::is_same<T, bool>::value &&
17     !std::is_same<T, std::string>::value &&
18     !std::is_same<T, const char*>::value &&
19     !std::is_same<T, char*>::value &&
20     !std::is_same<T, char>::value &&
21     !HasDefaultIOTraits<T>::value
22     >::type
23 {
24     ost << b;
25 }
26
27 // Overload for the argument of type bool
28 static void exportVariable(std::ostream& ost, const bool& b)
29 {
30     ost << (b == true ? "true" : "false");
31 }
32
33 // Overload of exportVariable for iterable list argument (e.g. std::map, std::vector)
34 template<typename T>
35 static auto exportVariable(std::ostream& ost, const T &list)
36     -> typename std::enable_if<
37     IsIterable<T>::value &&
38     !IsExportable<T>::value &&
39     !HasDefaultIOTraits<T>::value
40     >::type
41 {
42     auto it = list.cbegin();
43     ost << "[";
44     while (it != list.cend()){
45         exportVariable(ost, *it);
46         if (++it != list.cend()){
47             ost << ", ";
48         }
49     }
50     ost << "]";
51 }
52 // ** ... other overloads of exportVariable **
53 };

```

Code listing 4.4 The implementation of the function overload `variableExport` for the argument with `DefaultIOTraits` defined. This function utilizes a helper class `PrintClass` realizing the iteration over all traited attributes of the given instance of the traited class. The `print` member function of the general definition of `PrintClass` realizes the loop for `Index` from 0 to the number of traited attributes of the traited class minus one. In every iteration, it calls `print` of the specialization for the third template parameter `Print` equal to `true`, writes a comma, and recursively calls `print` of `PrintClass` with the `Index` increased by one. The `print` member function of the specialization exports the formatted message correspondingly to the `Index` into the given `std::ostream`. Thanks to the default value of the `Print` template parameter, when `Index` reaches the number of traited attributes minus one, the specialization is applied again and the recursion ends.

```

1 // ** inside VariableExport **
2 template< typename T,
3     unsigned int Index = 0,
4     bool Print = Index == DefaultIOTraits<T>::size() - 1>
5 struct PrintClass{
6     static void print(std::ostream& ost, const T &traitedClass){
7         PrintClass<T, Index, true>::print(ost, traitedClass);
8         ost << ",_";
9         PrintClass<T, Index + 1>::print(ost, traitedClass);
10
11     }
12 };
13
14 // Specialization terminating the recursion for the last traited attribute
15 template<typename T, unsigned int Index>
16 struct PrintClass<T, Index, true>{
17     static void print(std::ostream& ost, const T &traitedClass){
18         ost << ',' << DefaultIOTraits<T>::getTraits().template getName<Index>() << "\"_:_\"";
19         // Export the value into the stream
20         VariableExport::exportVariable(
21             ost,
22             DefaultIOTraits<T>::getTraits().template getValue<Index>(traitedClass)
23         );
24     }
25 };
26
27
28 template<typename T>
29 static auto exportVariable(std::ostream& ost, const T &traitedClass)
30     -> typename std::enable_if<
31         HasDefaultIOTraits<T>::value
32     >::type
33 {
34     ost << "{_";
35     PrintClass<T>::print(ost, traitedClass);
36     ost << "_}";
37 }

```

Code listing 4.5 The definition of the `writeVar` member function of the `ConsoleLogger` class. Here there are two overloaded definitions of `writeVar`. The `writeVar` functions together recursively perform the export of the given parameters. Both the functions accept the `line` and `cppFile` parameters. Furthermore, both functions accept the name and value of the exported variable. The second version of `writeVar` then accepts a variable number of additional arguments (`rest`). The first version of `writeVar` applies when exactly 4 arguments are passed and exports the formatted message to the standard error stream. The second version handles the case when extra arguments are passed. This `writeVar` passes the first four arguments (`line`, `cppFile`, `name` and `value` of the exported variable) and passes them to the first overload of `writeVar`. Then it calls the `writeVar` with `line`, `cppFile`, and the `rest` of the passed parameters. This recursion continues until the `rest` contains only 2 arguments, then the first overload is called again instead of the second version and the recursion stops.

```

1  template <VARIABLE_EXPORT_METHOD method = VARIABLE_EXPORT_METHOD::ostream>
2  class ConsoleLogger {
3
4  template<typename VAR>
5  static void writeVar(int line,
6                      const char* cppFile,
7                      const char* name,
8                      const VAR& value){
9
10     std::cerr << "==" << cppFile << "<<" << line << ">>[[<" << name << "]]_==>";
11     VariableExport<>::exportVariable(std::cerr, value);
12     std::cerr << "\n";
13
14 }
15
16 template<typename VAR, typename ... REST>
17 static void writeVar(int line,
18                     const char* cppFile,
19                     const const char*& name,
20                     const VAR& value,
21                     const REST& ... rest){
22
23     writeVar(line, cppFile, name, value);
24     writeVar(line, cppFile, rest...);
25 }
26
27 // ... ** other static member functions **
28 };

```


Code listing 4.6 The definition of the `DBGVAR` variadic macro. This macro accepts up to 40 expressions (see 3.15), which are processed by the `FOR_EACH` macro together with the `STRVAR` macro. As a result, `DBGVAR` expands into the call of `writeVar` of `ConsoleLogger` with the given line and file name where the macro is used. After the file name, a sequence of stringized expression and the expressions themselves corresponding to the expressions passed to `DBGVAR` follows. The last macro `DBGVAR_COND` simplifies the conditional export. It expects a condition as the first parameter. Then the message is printed only if the condition is satisfied.

```

1 // Macro expanding into stringized expression and the expression itself
2 #define STRVAR(var) #var, var
3
4 // Macro logging the contained expressions
5 #define DBGVAR(...) \
6     ConsoleLogger<>::writeVar(__LINE__, __FILE__, FOR_EACH(STRVAR, __VA_ARGS__))
7 #define DBGVARCOND(condition, ...) if(condition) DBGVAR(__VA_ARGS__)

```

The most frequently used debugger is the one with output to console. This functionality is provided by the `ConsoleLogger` class. `ConsoleLogger` has two main public static member functions, `writeMessage` and `writeVar`. Both the functions accept a variable number of parameters, where the first two are the line number to be written in the export line, and the name of the source file the export originates from `cppFile`. The rest of the parameters is a sequence of *name-value* pairs where *name* is a *string* containing a C++ expression and *value* contains its result. See the definition in Code listing 4.5.

Because the use of `writeMessage` and `writeVar` is not comfortable to use, the macros `DBGMSG` and `DBGVAR` in `Debug/Debug.h` were developed. Both the macros are variadic and automatically fill the `line` and `cppFile` in the function call. Moreover, `DBGVAR` utilizes the macro `FOR_EACH` (see Section 3.3.3) to automatically pass the stringized expressions to the `writeVar` function. `DBGVAR` allows up to 40 expressions to be evaluated and exported. See the definition of `DBGVAR` in Code listing 4.6.

The main advantage of this system is the comfort of its use and that the user always knows what result was exported and where the export is located in the code. For better orientation in the debug log, it is possible to turn on colored output by defining the macro `CONSOLE_COLORED_OUTPUT` before including the `Debug/Debug.h` header.

From Code listings 4.3 and 4.5, it is obvious that both `VariableExport` and `ConsoleLogger` are class templates. For some purposes, it may not be possible to use `std::ostream`, e.g., while performing on GPU using CUDA framework. Therefore, by changing the method template parameter to `stdio`, `ConsoleLogger` and `VariableExport` utilize the function `printf` instead of `std::ostream`. This way, the same debugging system can be utilized in CUDA kernels [15].

Sometimes the user might want to export larger containers and/or perform a large number of exports. For this purpose, there is a possibility to export the data into a file. However, it would not be efficient to write the human readable log as the `ConsoleLogger` does. Instead, we decided to utilize the JSON format. This way, the output data can be easily processed and analyzed, for example, using Python. This functionality is provided by the `JSONLogger` class. The only difference from the `ConsoleLogger` is that the `JSONLogger` does not have any global stream to write the output to. Therefore, the `JSONLogger` has a member `ofile` of type `std::ofstream` providing the access to the file. `JSONLogger` opens the file at the first write and closes the file at destruction of the class. The problem is the definition of a one static global instance in `Debug/Debug.h`. For this purpose, we utilized a `Singleton` class to ensure that only one instance of a class is created and repeated access obtains the one global instance. Then, the file is not reopened during every export.

Similarly to `ConsoleLogger`, the usage of `JSONLogger` is simplified a variadic macro `DBGVAR_JSON`. This macro utilizes the global instance of `JSONLogger` exporting into file named by `DBG.json` located in the working directory of the program. The example of the `JSONLogger`

Code listing 4.7 An example of the usage of the export of any expressions into the json file. The upper listing presents a sample code utilizing the debug export into json file. Notice that the usage of the `DBGVAR_JSON` does not differ from the usage of `DBGVAR` at all. The expressions passed to the macro are exported to a file with a structure shown in the lower listing. The exported file consists of a sequence of logs. Every log have its group index (`gInd`), specifying the logs generated by each single `JSONLogger` (`DBGVAR_JSON`) call, name of the source file (`file`), the line at which the macro was invoked, the expression written in the macro (`expr`) and the value of the expression (`data`).

```
1 #include "[[path]]/src/Debug/Debug.h"
2 #include <vector>
3 #include <list>
4 using namespace std;
5
6 int main() {
7     bool b = false;
8     std::list<int> list = {1,2,3};
9     std::vector<std::list<int>> vec(5, list);
10
11     DBGVAR_JSON(b, vec);
12     DBGVAR_JSON(!b, vec[0]);
13 }
```

```
1 {
2     "logs": [
3         {
4             "gInd" : 0,
5             "file" : "main.cpp",
6             "line" : 11,
7             "expr" : "b",
8             "data" : false
9         },
10        {
11            "gInd" : 0,
12            "file" : "main.cpp",
13            "line" : 11,
14            "expr" : "vec",
15            "data" : [ [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ], [ 1, 2, 3 ] ]
16        },
17        {
18            "gInd" : 1,
19            "file" : "main.cpp",
20            "line" : 12,
21            "expr" : "!b",
22            "data" : true
23        },
24        {
25            "gInd" : 1,
26            "file" : "main.cpp",
27            "line" : 12,
28            "expr" : "vec[0]",
29            "data" : [ 1, 2, 3 ]
30        }
31    ]
32 }
```

Code listing 4.8 An example of the traits algorithm usage. In other words, the traits algorithm provides arithmetic operators and mathematical functions accepting instances of objects with default arithmetic traits defined. The first listing presents an example of `cpp` file and the second presents its output.

```

1 #include "Debug/Debug.h"
2 #include "Traits/TraitsAlgorithm/TraitsAlgorithm.h"
3 class Data {
4 public:
5     double D;
6     int I;
7     // If the attributes are private, declare the Traits as a friend
8     friend DefaultArithmeticTraits<Data>;
9 };
10
11 // Create specialization of DefaultArithmeticTraits for the class Data
12 MAKE_ATTRIBUTE_TRAIT_ARITHMETIC(Data, D, I);
13 // Create specialization of DefaultIOTraits for the class Data
14 MAKE_NAMED_ATTRIBUTE_TRAIT_IO(Data, "double_attr", D, "int_attr", I);
15
16 int main() {
17     Data d1{12.5, 68}, d2{14.0, 22};
18     DBGVAR(d1+d2, d1 * 2, max(d1), pow(d1, 2));
19 }

```

```

1 == main.cpp << 17 >> [[ d1+d2 ]] ==> { "double attr" : 26.5, "int attr" : 90 }
2 == main.cpp << 17 >> [[ d1 * 2 ]] ==> { "double attr" : 25, "int attr" : 136 }
3 == main.cpp << 17 >> [[ max(d1) ]] ==> 68
4 == main.cpp << 17 >> [[ pow(d1, 2) ]] ==> { "double attr" : 156.25, "int attr" : 4624 }

```

usage and output is in Code listing 4.7.

4.2 Numerical Algorithms Based on Class Traits

In the description of the concept of class traits (Chapter 3) we discussed that this concept underwent a series of optimizations. The aim of the optimizations was to develop class member access mechanism as efficient as explicitly written code. Thanks to the optimizations, it is possible to engage the class traits in numerical computation, e.g., addition or multiplication, without an impact on performance. All the operations then work with the traited classes as a vectors and the operations are defined element-wise.

The operations are divided into three types:

- Binary: accepts two arguments and returns a new instance of the traited class, e.g., sum of two traited objects or multiplication of a traited class by a number.
- Unary: accepts one traited class and returns a traited class, e.g., absolute value.
- Aggregation: accepts one argument of traited class and returns a value, e.g., maximum accepts an instance of a traited class and returns one number (the greatest element).

The functions and operators providing arithmetic operations on traited classes are defined in the header file `Traits/TraitsAlgorithm/TraitsAlgorithm.h`.

Let us continue with the implementation details of the arithmetic operations. Recall that a similar concept was already engaged in the definition of arithmetic operations on the `Vertex` class, see Code listings 2.10 and 2.11. Especially, the following solution extends the technique presented in Code listing 2.11.

We start with the description of the implementation of unary operations because it has the most straightforward implementation of the three cases. The functionality is provided by the class template `TraitsUnaryExpressionProcessor`. This class has one template parameter:

Code listing 4.9 The implementation of `TraitsUnaryExpressionProcessor`. This class accepts a class template `Operator` with one template parameter. The `Operator` class must have a public static member function `Operator::evaluate`. `TraitsUnaryExpressionProcessor` has two overloads of the `evaluate` member function. Both `evaluate` functions accept two instances of a traited class. The first is named `res` and holds the result. The second is the operand to be processed `op1`. The functions together apply `Operator::evaluate` to each traited attribute of the given traited class by using template recursion similarly to `PrintClass` (see Code listing 4.4) with the difference that the applied function overload is resolved in SFINAE context (invoked by the return type) of the `evaluate` functions instead of the template specialization.

```

1  template<template<typename> class Operator>
2  struct TraitsUnaryExpressionProcessor {
3
4      template<typename TraitT, unsigned int Index = 0,
5              bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
6      inline static
7      typename std::enable_if<ApplyOperation>::type
8      evaluate(TraitT& res, const TraitT& op1){
9
10         DefaultArithmeticTraits<TraitT>::getTraits().template getAttr<Index>(res) =
11         Operator<
12             typename DefaultArithmeticTraits<TraitT>::traitsType::template type<Index>
13         >::evaluate(
14             DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1)
15         );
16     }
17
18     template<typename TraitT, unsigned int Index = 0,
19             bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
20     inline static
21     typename std::enable_if<!ApplyOperation>::type
22     evaluate(TraitT& res, const TraitT& op1){
23
24         TraitsUnaryExpressionProcessor<Operator>::evaluate<TraitT, Index, true>(res, op1);
25         TraitsUnaryExpressionProcessor<Operator>::evaluate<TraitT, Index + 1>(res, op1);
26     }
27 };

```

Code listing 4.10 An example of an unary operator. The `Abs` class template has one member function named `evaluate`. The `evaluate` function accepts a single parameter of the type passed to the `Abs` class as the template parameter. The `evaluate` function then calls the `abs` function on the passed attribute. Then, the `Abs` class template is utilized by the following `abs` functions. Both the `abs` overloads check whether the passed parameter has the default arithmetic traits defined in SFINAE context. Then, both functions utilize the `TraitsUnaryExpressionProcessor` to apply the `Abs` operation element-wise. The second listed `abs` function accepts an r-value reference and stores the result directly in the passed instance. This may improve performance by preventing the call of the constructor of the traited class.

```

1  template< typename T1 >
2  struct Abs
3  {
4      static auto evaluate( const T1& a ) -> decltype(abs(a)) {
5          return abs(a);
6      }
7  };
8
9  template <typename TraitT>
10 typename std::enable_if<HasDefaultArithmeticTraits<TraitT>::value, TraitT>::type
11 abs(const TraitT& op1) noexcept {
12
13     TraitT res;
14     TraitsUnaryExpressionProcessor<Abs>::evaluate(res, op1);
15     return res;
16 }
17
18 template <typename TraitT>
19 typename std::enable_if<HasDefaultArithmeticTraits<TraitT>::value, TraitT>::type
20 abs(TraitT&& op1) noexcept {
21
22     TraitsUnaryExpressionProcessor<Abs>::evaluate(op1, op1);
23     return op1;
24 }

```

- **Operator** (`template<typename> class`): class with one template argument.

Additionally, the given type `Operator` must have public static member function `evaluate` defined. `TraitsUnaryExpressionProcessor` has two overloads of `evaluate` static member function template. The template parameters of `TraitsUnaryExpressionProcessor::evaluate` are:

- **TraitT** (`typename`): the type of accepted traited class (deduced from the type of passed parameters).
- **Index** (`unsigned int`): the index of currently processed traited attribute 0 by default.
- **applyOperation** (`bool`): specifies what overload shall be applied (discussed below). By default, it is true if the `Index` points to the last traited attribute of the `TraitT` class, false otherwise.

See the implementation in Code listing 4.9.

The usage of `TraitsUnaryExpressionProcessor` is the following. At first, prepare the class to be passed as `Operator`. We demonstrate the problem on an example of a function calculating absolute value, i.e., `abs`. Therefore the class to be passed as `Operator` is named `Abs` and has the `evaluate` member function which calls `abs`. See Code listing 4.10. The next step is to create the `abs` function template for classes with `DefaultArithmeticTraits` defined. The check is done in SFINAE context, where the type trait `HasDefaultArithmeticTraits` (see Section 3.3.1) is utilized. The `abs` function for the traited class then utilizes the prepared processor class given `Abs` as `Operator`. Moreover, there is an optimization overload of the `abs` function accepting an r-value instead of l-value reference. The optimization consists in storing the result directly in the

passed argument instead of creating a new instance of the traited class. See the implementation in Code listing 4.10.

The implementation of binary operations is very similar to the unary ones. Binary operations utilize the class `TraitsBinaryExpressionProcessor`. Unlike the unary processor class, this class has four overloads of the `evaluate` member function and the template parameter `Operator` accept two template parameters instead of one. All of the overloads accepts three arguments, where the first two are the references to the traited class instances. The first is a container for the result and the second is the first operand. Then the overloads differ in the type of the third argument. The first two functions expect the third argument of traited class type and apply the given operation to all traited attributes. This is applied, for example, in the implementation of element-wise addition of two instances of a traited class. The remaining two overloads expect a different type from the given traited class, e.g., a number. These functions apply the given operation for each traited attribute with the same second operand, e.g., the number. This approach is utilized, for example, in implementation of multiplication of a traited class by a number. The implementation of the final operators is realized in the same manner as in the case of the `abs` function discussed above. The implementation of `TraitsBinaryExpressionProcessor` is in Code listing 4.11.

The last type of operations are aggregation operations, e.g., sum or maximum. Let us note that this concept is new in GTMesh and may be not optimal. Similarly to the previous two cases, there is a class template `TraitsAggregationProcessor` realizing the compile time loop unrolling. Similarly to the binary case, this class expects a binary operator. Then, it uses recursive operator evaluation where the previous result is used as the second operand.

Let us describe the calculation of an aggregation operation on a simple example. Suppose a traited class with 3 attributes of type `double`. At first, the calculation function applies the aggregation function on the first two attributes. Then, the function is applied again to the previous result and the third attribute. Eventually, the result of this operation is the expected aggregated result.

In the concept of class traits, we have not put any restrictions on the types of the traited attributes. Therefore, it is possible to have one double attribute and another `Vertex<3, double>`. In such case, we challenge the problem of comparing a number with a `Vertex` class. However, we expect the class to behave as a compact vector without any inner structure. The solution is to create two more overloads of the `evaluate` member function of `TraitsAggregationProcessor`. The first overload is for non-class arguments, e.g., `int` or `double` and it returns the value of the given argument. The second overload accepts the indexable arguments (with `operator[]` defined). This function utilizes `Operator` to calculate the aggregated value of the array arguments. These two overloads are called inside the loop unrolling `evaluate` function. Additionally, let us mention that the return type of the evaluate function is not a priori known. This can be solved by the `auto` return type, i.e., the return type is deduced from the return statements. Therefore, to utilize the `auto` return type, the SFINAE is located in the template parameters (type of the last unnamed parameter). See the implementation in Code listing 4.12.

Code listing 4.11 The implementation of the `TraitsBinaryExpressionProcessor` class template. This class utilizes a concept similar to `TraitsUnaryExpressionProcessor` to perform a binary operation element-wise. Additionally, this case is extended by two more overloads of the `evaluate` function. These two variants of `evaluate` are designed to apply the binary operation to a traited class and a real number, as in the case of multiplication of all attributes by a number.

```

1  template<template<typename, typename> class Operator>
2  struct TraitsBinaryExpressionProcessor {
3
4  template<typename TraitT, unsigned int Index = 0,
5          bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
6  inline static
7  typename std::enable_if<!ApplyOperation>::type
8  evaluate(TraitT& res, const TraitT& op1, const TraitT& op2){
9
10     TraitsBinaryExpressionProcessor<Operator>::evaluate<TraitT, Index, true>(res, op1, op2);
11     TraitsBinaryExpressionProcessor<Operator>::evaluate<TraitT, Index + 1>(res, op1, op2);
12 }
13
14 template<typename TraitT, unsigned int Index = 0,
15         bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
16 inline static
17 typename std::enable_if<ApplyOperation>::type
18 evaluate(TraitT&res, const TraitT& op1, const TraitT& op2){
19
20     DefaultArithmeticTraits<TraitT>::getTraits().template getAttr<Index>(res) =
21         Operator<
22             typename DefaultArithmeticTraits<TraitT>::traitsType::template type<Index>,
23             typename DefaultArithmeticTraits<TraitT>::traitsType::template type<Index>
24         >::evaluate(
25             DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1),
26             DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op2)
27         );
28 }
29
30 template<typename TraitT, typename Real, unsigned int Index = 0,
31         bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
32 inline static
33 typename std::enable_if<!ApplyOperation>::type
34 evaluate(TraitT& res, const TraitT& op1, const Real& op2){
35
36     TraitsBinaryExpressionProcessor<Operator>::
37         evaluate<TraitT, Real, Index, true>(res, op1, op2);
38     TraitsBinaryExpressionProcessor<Operator>::
39         evaluate<TraitT, Real, Index + 1>(res, op1, op2);
40 }
41
42 template<typename TraitT, typename Real, unsigned int Index = 0,
43         bool ApplyOperation = Index == DefaultArithmeticTraits<TraitT>::size() - 1>
44 inline static
45 typename std::enable_if<ApplyOperation>::type
46 evaluate(TraitT&res, const TraitT& op1, const Real& op2){
47
48     DefaultArithmeticTraits<TraitT>::getTraits().template getAttr<Index>(res) =
49         Operator<
50             typename DefaultArithmeticTraits<TraitT>::traitsType::template type<Index>,
51             Real
52         >::evaluate(
53             DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1),
54             op2
55         );
56 }
57 };

```

Code listing 4.12 The implementation of `TraitsAggregationProcessor`. This class is responsible for realization of the aggregation process. In the current implementation, the aggregation process utilizes a binary operator, e.g., plus or maximum. This operator is then applied to an attribute and previously calculated result provided by the first two overloads of the `evaluate` member function. However, the traited class is not limited to having only attributes of simple types (e.g., `Vector`). Therefore, we decided to create two more overloads of the `evaluate` function. The third overload is applied in the case that the given parameter is a vector (indexable object). In such case, the aggregated value of the vector is utilized. The last overload prevents a compilation error when applying the `evaluate` function on a simple type, e.g., `double`.

```

1  template<typename, typename> class Operator>
2  struct TraitsAggregationProcessor {
3
4      template< typename TraitT,
5                unsigned int Index = DefaultArithmeticTraits<TraitT>::size() - 1,
6                typename std::enable_if< Index == 0, bool >::type = true >
7      inline static
8      auto
9      evaluate(const TraitT& op1){
10         return evaluate(DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1));
11     }
12
13     template< typename TraitT,
14              unsigned int Index = DefaultArithmeticTraits<TraitT>::size() - 1,
15              typename std::enable_if<(Index > 0) &&
16              (Index <= DefaultArithmeticTraits<TraitT>::size() - 1), bool >::type = true>
17     inline static
18     auto
19     evaluate(const TraitT& op1){
20
21         return Operator<
22             decltype(evaluate<TraitT, Index - 1>(op1)),
23             decltype(evaluate(DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1)))
24             >::evaluate(
25                 (evaluate<TraitT, Index - 1>(op1)),
26                 (evaluate(DefaultArithmeticTraits<TraitT>::getTraits().template getValue<Index>(op1)))
27             );
28     }
29
30
31     template <typename T, typename std::enable_if< !std::is_class<T>::value, bool >::type = true>
32     inline static
33     auto
34     evaluate(const T& arg) noexcept {
35         return arg;
36     }
37
38     template <typename T, typename std::enable_if< IsIndexable<T>::value, bool >::type = true>
39     inline static
40     auto
41     evaluate(const T& array) noexcept {
42         if (array.size() > 0){
43             using resType = decltype (evaluate(array[0]));
44             auto res = evaluate(array[0]);
45             for (decltype (array.size()) index = 1; index < array.size(); index++){
46                 res = Operator<resType, resType>::evaluate(res, evaluate(array[index]));
47             }
48             return res;
49         }
50         return resType();
51     }
52 };

```


4.3 The Runge-Kutta-Merson Solver

Thanks to the concept of traits algorithm (see Section 4.2), it is possible to create general numeric functions. In this case, we utilized the traits algorithm to implement the Merson version of Runge-Kutta explicit solver (RKM) [33] for ordinary differential equations in the form

$$\dot{\mathbf{x}} = f(t, \mathbf{x}), \quad (4.1)$$

where the $\mathbf{x} \in \mathbb{R}^N$, $N \in \mathbb{N}$ and $f: \mathcal{I} \times \mathbb{R}^N \rightarrow \mathbb{R}^N$.

Firstly, we begin with the description of the algorithm. Let us firstly introduce the parameters of the method.

t	current time level
T	final time
τ	time step
τ_{ini}	initial time step
\mathbf{x}^τ	numerical solution
$\mathbf{x}_{\text{ini}}^\tau$	initial condition for the numerical solution \mathbf{x}^τ
δ	tolerance parameter

The modified version of the 4th order accurate Runge-Kutta-Merson solver for numerical time integration of is presented by the following pseudo-code:

```

1   $\tau = \tau_{\text{ini}}; \mathbf{x}^\tau = \mathbf{x}_{\text{ini}}^\tau;$ 
2  while(true){
3      last = false;
4      if ( |T - t| < |t| ) {
5           $\tau = T - t;$ 
6          last = true;
7      }
8       $\mathbf{K}_1 = f(t, \mathbf{x}^\tau);$ 
9       $\mathbf{K}_2 = f(t + \frac{\tau}{3}, \mathbf{x}^\tau + \frac{\tau}{3}\mathbf{K}_1);$ 
10      $\mathbf{K}_3 = f(t + \frac{\tau}{3}, \mathbf{x}^\tau + \frac{\tau}{6}(\mathbf{K}_1 + \mathbf{K}_2));$ 
11      $\mathbf{K}_4 = f(t + \frac{\tau}{2}, \mathbf{x}^\tau + \frac{\tau}{8}(\mathbf{K}_1 + 3\mathbf{K}_3));$ 
12      $\mathbf{K}_5 = f(t + \tau, \mathbf{x}^\tau + \tau(\frac{1}{2}\mathbf{K}_1 - \frac{3}{2}\mathbf{K}_3 + 2\mathbf{K}_4));$ 
13      $\varepsilon = \max_{\text{element}} \frac{\tau}{3} |0.2\mathbf{K}_1 - 0.9\mathbf{K}_3 + 0.8\mathbf{K}_4 - 0.1\mathbf{K}_5|$ 
14     if (  $\varepsilon < \tau$  ) {
15          $\mathbf{x}^\tau = \mathbf{x}^\tau + \tau(\frac{1}{6}(\mathbf{K}_1 + \mathbf{K}_5) + \frac{2}{3}\mathbf{K}_4);$ 
16          $t = t + \tau;$ 
17         if (last) break;
18         if (  $\varepsilon == 0$  ) continue;
19     }
20      $\tau = (\delta/\varepsilon)^{0.2} \cdot \omega\tau;$ 
21 }
```

In this work, the value of ω is fixed to 0.8. For more information, see [33, 35, 49]. This algorithm was utilized to calculate the numerical solution of the multi-phase flow problem (see Chapter 5).

The algorithm was implemented as the `RKMSolver` function template accepting the type `Problem`, e.g., `MultiphaseFlow` described in Section 5.3. In the current implementation, there are several assumptions on the `Problem` class. This class must provide public alias names for the data type the function works with and the utilized mesh. The `RKMSolver` utilizes the `GTMesh` concepts, especially `MeshDataContainer` as the container for the numerical data. The implemented algorithm runs in parallel using OpenMP [18].

Thanks to the traits algorithm, the RKM solver can be applied to any class with default arithmetic traits defined. Using custom structures to represent numerical solution data is very convenient for the user and this implementation of the RKM solver is a suitable component to

the generic concept of `MeshDataContainer` (see Section 2.3). Moreover, the tools exporting and importing data from the files are compatible with class traits and significantly reduce the user's effort during implementation of numerical solvers (see Section 5.3).

Finally, let us note that the developed numerical solver is more like proof of concept than a complete solution.

Chapter 5

Numerical Solution of Compressible Two-phase Flow

In this chapter, the application of GTMesh on the numerical solution of two-phase flow will be presented. The problem in question stems from the study of the processes in the combustion chamber of fluidized bed boilers [25] where fluidization [53] of a granular material (a mixture of limestone and fuel particles) is achieved by a strong enough air flow. The mathematical formulation of the two-phase flow is based on the continuum (Eulerian) approach [36, 38] where the conservation laws for both phases are formulated in the form of partial differential equations. The Resulting problem is subsequently solved by using the finite volume method [30] (FVM) on general unstructured meshes in 2D and 3D and the Runge-Kutta-Merson time integrator (see Section 4.3).

5.1 Governing Equations

Let $\Omega \subset \mathbb{R}^d$ where $d \in \{2, 3\}$ be a domain and $\mathcal{J} = (0, T_{\text{end}})$ time interval, where T_{end} is the final time.

Firstly, let us recall the mass and momentum conservation laws for a Newtonian fluid [36, 42, 45, 53].

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u}), \quad (5.1)$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla p + \nabla \cdot \mathbb{T}, \quad (5.2)$$

where $\rho : \Omega \times \mathcal{J} \rightarrow \mathbb{R}^+$ is the density, $\mathbf{u} : \Omega \times \mathcal{J} \rightarrow \mathbb{R}^d$ is the velocity, $p : \Omega \times \mathcal{J} \rightarrow \mathbb{R}$ is the pressure and $\otimes : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^{d,d}$, satisfying for $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ denotes the tensor product

$$(\mathbf{u} \otimes \mathbf{v})_{kl} = u_k v_l, \quad (5.3)$$

where $k, l \in \{1, \dots, d\}$. Finally, $\mathbb{T} : \Omega \times \mathcal{J} \rightarrow \mathbb{R}^{d \times d}$ is the viscous stress tensor, which assumes the following form

$$\mathbb{T} = \tau_{kl} = \mu \left(\frac{\partial u_l}{\partial x_k} + \frac{\partial u_k}{\partial x_l} \right) - \delta_{kl} \frac{2\mu}{3} \nabla \cdot \mathbf{u}. \quad (5.4)$$

The divergence of the stress tensor is defined as

$$\nabla \cdot \mathbb{T} = \nabla \cdot (\boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \dots, \boldsymbol{\tau}_d), \quad (5.5)$$

where $\boldsymbol{\tau}_i$ is the i th column of the tensor \mathbb{T} , $i \in \{1, \dots, d\}$.

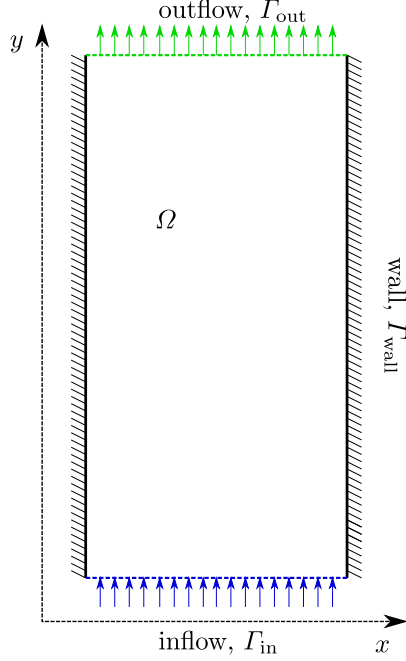


Figure 5.1: Simple example of the problem setup

Finally, equations for both phases are similar to each other and both are based on equations (5.1) and (5.2). Because there are two phases, gaseous and solid, the quantities are distinguished by the index g for gaseous state and s for solid state. Then, the problem of two-phase flow reads

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho_g \varepsilon_g \\ \rho_g \varepsilon_g \mathbf{u}_g \\ \rho_s \varepsilon_s \\ \rho_s \varepsilon_s \mathbf{u}_s \end{pmatrix} + \begin{pmatrix} \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g) \\ \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g \otimes \mathbf{u}_g) \\ \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s) \\ \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s \otimes \mathbf{u}_s) \end{pmatrix} = \begin{pmatrix} 0 \\ -\varepsilon_g \nabla p_g + \nabla \cdot (\varepsilon_g \mathbb{T}_g) \\ 0 \\ -G(\varepsilon_g) \nabla \varepsilon_s - \varepsilon_s \nabla p_g + \nabla \cdot (\varepsilon_s \mathbb{T}_s) \end{pmatrix} + \begin{pmatrix} 0 \\ \rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g) \\ 0 \\ (\rho_s - \rho_g) \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s) \end{pmatrix}, \quad (5.6)$$

posed in $\Omega \times \mathcal{I}$, where the unknown quantities are described in Table 5.1. The term $G(\varepsilon_g) \nabla \varepsilon_s$ plays a role analogous to ∇p_s , to prevent spatial collapsing of the solid phase. The function G is called modulus of compressibility [36] and reads

$$G(\varepsilon_g) = 10^{-8.76\varepsilon_g + 5.43}. \quad (5.7)$$

The coefficient β_{gs} represents momentum transfer between the gaseous and solid phases (drag). The form of β_{gs} used in (5.6) reads [36]

$$\beta_{gs} = \begin{cases} 150 \frac{\varepsilon_s^2 \mu_g}{(\varepsilon_g d_s \Phi_s)^2} + 1.75 \frac{|\mathbf{u}_g - \mathbf{u}_s| \rho_g \varepsilon_s}{\varepsilon_g d_s \Phi_s} & \varepsilon_s > 0.2, \\ \frac{4}{3} C_d \frac{|\mathbf{u}_g - \mathbf{u}_s| \rho_g \varepsilon_s}{d_s \Phi_s} & \varepsilon_s \leq 0.2, \end{cases} \quad (5.8)$$

$$C_d = \begin{cases} \frac{24}{Re_s} (1 + 0.15 Re_s^{0.687}) & Re_s \leq 1000, \\ 0.44 & Re_s > 1000, \end{cases} \quad (5.9)$$

$$Re_s = \frac{|\mathbf{u}_g - \mathbf{u}_s| d_s \rho_g \varepsilon_g}{\mu_g}. \quad (5.10)$$

quantity	unit	range	description
p_g	Pa	\mathbb{R}	pressure of the gaseous phase
T	K	\mathbb{R}^+	temperature [constant]
ρ_g	$\text{kg} \cdot \text{m}^{-3}$	\mathbb{R}^+	density of the gaseous phase
\mathbf{u}_g	$\text{m} \cdot \text{s}^{-1}$	\mathbb{R}^3	velocity of the gaseous phase
ε_g	-	$[0, 1]$	volume fraction of the gaseous phase
ρ_s	$\text{kg} \cdot \text{m}^{-3}$	\mathbb{R}^+	solid density [constant]
\mathbf{u}_s	$\text{m} \cdot \text{s}^{-1}$	\mathbb{R}^3	velocity of the solid phase
ε_s	-	$[0, 1]$	volume fraction of the solid phase

Table 5.1: List of quantities used in (5.6).

The symbol \mathbf{g} denotes gravitational acceleration, d_s is the diameter of the solid state particles and ϕ_s is their sphericity [43].

The additional relations

$$\varepsilon_g + \varepsilon_s = 1, \quad (5.11)$$

$$p_g = \rho_g R_{\text{spec}} T \quad (5.12)$$

close the system. In (5.12), R_{spec} is specific the gas constant and T stands for temperature, which is the same for both phases and remains constant.

In the end, the system consists of 8 equations for 8 unknown quantities. For the full list of quantities see Table 5.1.

5.1.1 Initial Conditions

The initial conditions for the two-phase flow problem read

$$\begin{aligned} p_g(\mathbf{x}, 0) &= p_{g,\text{ini}}(\mathbf{x}), \\ \mathbf{u}_g(\mathbf{x}, 0) &= \mathbf{u}_{g,\text{ini}}(\mathbf{x}), \\ \varepsilon_s(\mathbf{x}, 0) &= \varepsilon_{s,\text{ini}}(\mathbf{x}), \\ \mathbf{u}_s(\mathbf{x}, 0) &= \mathbf{u}_{s,\text{ini}}(\mathbf{x}). \end{aligned} \quad (5.13)$$

The rest of the quantities are calculated as follows:

$$\begin{aligned} \rho_g(\mathbf{x}, 0) &= \frac{p_{g,\text{ini}}(\mathbf{x})}{R_{\text{spec}} T}, \\ \varepsilon_g(\mathbf{x}, 0) &= 1 - \varepsilon_{s,\text{ini}}(\mathbf{x}). \end{aligned} \quad (5.14)$$

5.1.2 Boundary Conditions

There are three types of boundary conditions considered: inlet, outlet and wall. According to the applied boundary condition, the boundary of the spatial domain is divided into Γ_{in} , Γ_{out} , Γ_{wall} satisfying $\partial\Omega = \Gamma_{\text{in}} \cup \Gamma_{\text{out}} \cup \Gamma_{\text{wall}}$. Thanks to relations (5.11), (5.12) and the fact that the temperature T and the solid density ρ_s remain constant, it is only needed to prescribe the boundary conditions for ρ_g , \mathbf{u}_s , \mathbf{u}_g , ε_s .

- For $\mathbf{x} \in \Gamma_{\text{wall}}$, the boundary conditions consist of the non-slip boundary condition for velocities, i.e.

$$\mathbf{u}_s(\mathbf{x}, t) = \mathbf{u}_g(\mathbf{x}, t) = 0 \quad \forall t \in \mathcal{I}, \quad (5.15)$$

and the zero Neumann boundary condition applied to the gas density and the volume fraction of solid phase:

$$\frac{\partial \rho_g(\mathbf{x}, t)}{\partial \mathbf{n}} = \frac{\partial \varepsilon_s(\mathbf{x}, t)}{\partial \mathbf{n}} = 0 \quad \forall t \in \mathcal{I}, \quad (5.16)$$

where \mathbf{n} is the outward normal vector of $\partial\Omega$ at the point \mathbf{x} .

- For $\mathbf{x} \in \Gamma_{\text{in}}$, the value of velocity and solid volume fraction is prescribed:

$$\mathbf{u}_g(\mathbf{x}, t) = \mathbf{u}_{g,\text{in}}(\mathbf{x}, t), \quad (5.17)$$

$$\mathbf{u}_s(\mathbf{x}, t) = \mathbf{u}_{s,\text{in}}(\mathbf{x}, t), \quad (5.18)$$

$$\varepsilon_s(\mathbf{x}, t) = \varepsilon_{s,\text{in}}(\mathbf{x}, t). \quad (5.19)$$

Furthermore, a zero Neumann boundary condition for the density of gas is prescribed:

$$\frac{\partial \rho_g(\mathbf{x}, t)}{\partial \mathbf{n}} = 0. \quad (5.20)$$

- For $\mathbf{x} \in \Gamma_{\text{out}}$, the value of pressure is set:

$$p_g(\mathbf{x}, t) = p_{g,\text{out}}(\mathbf{x}, t). \quad (5.21)$$

For the rest of the quantities, zero Neumann conditions are imposed:

$$\frac{\partial \mathbf{u}_g(\mathbf{x}, t)}{\partial \mathbf{n}} = \frac{\partial \mathbf{u}_s(\mathbf{x}, t)}{\partial \mathbf{n}} = \vec{0} \quad (5.22)$$

$$\frac{\partial \varepsilon_s(\mathbf{x}, t)}{\partial \mathbf{n}} = 0 \quad (5.23)$$

5.2 Numerical Scheme

In this section a spatial discretization of the equations (5.6) will be derived. The numerical method used to obtain the semi-discrete scheme on general unstructured meshes is the finite volume method [30]. The resulting system of ODE is solved by the 4th order Runge-Kutta-Merson method with adaptive time stepping.

5.2.1 Unstructured Mesh Notation

The basic notations were already presented in Section 1.2, see Definition 1. However, for the purpose of developing the numerical schemes, we present some additional definitions and notations. The first defined mesh property is called admissibility.

Definition 8. The mesh \mathcal{T} is called admissible if the following conditions are met

1. $(\exists \mathcal{P} \subset \Omega) ((\forall K \in \mathcal{T}) (\exists \mathbf{x}_K \in \mathcal{P}) (\mathbf{x}_K \in \bar{K}) \wedge (\forall \mathbf{x} \in \mathcal{P}) (\exists K \in \mathcal{T}) (\mathbf{x} \in \bar{K}))$.
2. $(\forall K, L \in \mathcal{T}) (K \neq L \implies (\mathbf{x}_K \neq \mathbf{x}_L \wedge \overline{\mathbf{x}_K \mathbf{x}_L} \perp K|L))$. We denote by $\mathcal{D}_{K,L} = \overline{\mathbf{x}_K \mathbf{x}_L}$ the line connecting vertices \mathbf{x}_K and \mathbf{x}_L , which is perpendicular to the edge $K|L$.

Now some additional notations are to be introduced:

- Let $K \in \mathcal{T}$, $\sigma \in \mathcal{E}_K$ such that $\sigma \in \partial\Omega$. If $\mathbf{x}_K \notin \sigma$, then $\mathcal{D}_{K,\sigma} = \overline{\mathbf{x}_K \mathbf{y}_\sigma}$, while $\mathbf{y}_\sigma \in \sigma$ is such that $\mathcal{D}_{K,\sigma} \perp \sigma$ holds.
- $\mathcal{E}_{\text{ext}} = \{\sigma \in \mathcal{E} \mid \sigma \in \partial\Omega\}$, $\mathcal{E}_{\text{int}} = \mathcal{E} \setminus \mathcal{E}_{\text{ext}}$.
- For $\sigma \in K|L$, $d_\sigma = |\mathcal{D}_{K,L}|$ denotes the Euclidean distance between \mathbf{x}_K and \mathbf{x}_L . Similarly, the distance from \mathbf{x}_K to σ is denoted $d_{K,\sigma} = |\mathcal{D}_{K,\sigma}|$. For $\sigma \in \mathcal{E}_K \cap \mathcal{E}_{\text{ext}}$, we define $d_\sigma := d_{K,\sigma}$.
- Finally, the edge/face measure over cell centers distance is labeled $\tau_\sigma = m(\sigma)/d_\sigma$.

The above defined notation is demonstrated in Figure 5.2.

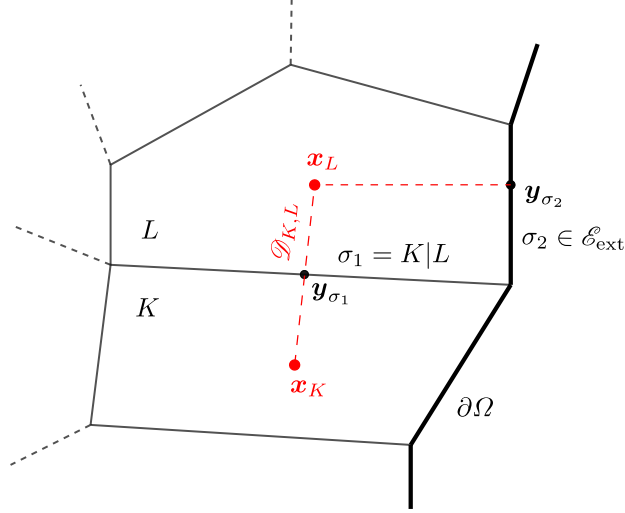


Figure 5.2: Notation of the finite volume method on an unstructured mesh.

5.2.2 Finite Volume Method on Unstructured Meshes

In this section, the finite volume method (FVM) will be applied to the system of equations (5.6). For easier understanding, the approximation using FVM will be performed separately for each equation. To improve numerical stability in some simulations, we add an artificial diffusion (dissipation) term to the right hand side of (5.6), so that it assumes the form

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho_g \varepsilon_g \\ \rho_g \varepsilon_g \mathbf{u}_g \\ \rho_s \varepsilon_s \\ \rho_s \varepsilon_s \mathbf{u}_s \end{pmatrix} + \begin{pmatrix} \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g) \\ \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g \otimes \mathbf{u}_g) \\ \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s) \\ \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s \otimes \mathbf{u}_s) \end{pmatrix} = \begin{pmatrix} 0 \\ -\varepsilon_g \nabla p_g + \nabla \cdot (\varepsilon_g \mathbb{T}_g) \\ 0 \\ -G(\varepsilon_g) \nabla \varepsilon_s - \varepsilon_s \nabla p_g + \nabla \cdot (\varepsilon_s \mathbb{T}_s) \end{pmatrix} + \begin{pmatrix} \lambda_g \Delta (\rho_g \varepsilon_g) \\ \lambda_g \Delta (\rho_g \varepsilon_g \mathbf{u}_g) \\ \lambda_s \Delta (\rho_s \varepsilon_s) \\ \lambda_s \Delta (\rho_s \varepsilon_s \mathbf{u}_s) \end{pmatrix} + \begin{pmatrix} 0 \\ \rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g) \\ 0 \\ (\rho_s - \rho_g) \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s) \end{pmatrix}, \quad (5.24)$$

where λ_g and λ_s are the coefficients of artificial diffusion.

We are using the cell-centered finite volume scheme, which is cheap to implement, with quantities evaluated in the centers of cells.

Note that the theory of convergence and error estimation of the numerical solution is complicated and is not the aim of this work. For more details see [30, 45].

Conservation of Mass of the Gaseous Phase

First, we integrate the first row of (5.24) over a control volume $K \in \mathcal{T}$ to obtain

$$\int_K \frac{\partial}{\partial t} (\rho_g \varepsilon_g) \, d\mathbf{x} = \int_K (-\nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g) + \lambda_g \Delta (\rho_g \varepsilon_g)) \, d\mathbf{x}. \quad (5.25)$$

Under the corresponding regularity assumptions, this relation can be equivalently rewritten using Gauss-Green theorem into the form

$$\int_K \frac{\partial}{\partial t} (\rho_g \varepsilon_g) \, d\mathbf{x} = - \oint_{\partial K} \rho_g \varepsilon_g (\mathbf{u}_g \cdot \mathbf{n}) \, dS + \oint_{\partial K} \lambda_g \frac{\partial (\rho_g \varepsilon_g)}{\partial \mathbf{n}} \, dS, \quad (5.26)$$

where \mathbf{n} is the outward pointing normal vector to ∂K . Thanks to the fact that $\partial K = \cup_{\sigma \in \mathcal{E}_K} \sigma$, the right hand side of equation (5.26) is equal to

$$\oint_{\partial K} \rho_g \varepsilon_g (\mathbf{u}_g \cdot \mathbf{n}) dS = \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \rho_g \varepsilon_g (\mathbf{u}_g \cdot \mathbf{n}) dS, \quad (5.27)$$

$$\oint_{\partial K} \lambda_g \frac{\partial (\rho_g \varepsilon_g)}{\partial \mathbf{n}} dS = \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \lambda_g \frac{\partial (\rho_g \varepsilon_g)}{\partial \mathbf{n}} dS. \quad (5.28)$$

Finally, Leibnitz integral rule is used on the left hand side. The resulting formula suitable for discretization is

$$\frac{d}{dt} \int_K (\rho_g \varepsilon_g) d\mathbf{x} = - \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \rho_g \varepsilon_g (\mathbf{u}_g \cdot \mathbf{n}) dS + \sum_{\sigma \in \mathcal{E}_K} \lambda_g \int_{\sigma} \frac{\partial (\rho_g \varepsilon_g)}{\partial \mathbf{n}} dS. \quad (5.29)$$

The next step is to approximate the integrals. We denote the numerical approximation of the solution of the first row of the problem (5.24) in the following manner:

$$\rho_g(\mathbf{x}_K, t) \approx \rho_{g,K}(t), \quad (5.30)$$

$$\varepsilon_g(\mathbf{x}_K, t) \approx \varepsilon_{g,K}(t), \quad (5.31)$$

$$\mathbf{u}_g(\mathbf{x}_K, t) \approx \mathbf{u}_{g,K}(t), \quad (5.32)$$

$$\rho_g(\mathbf{y}_{\sigma}, t) \approx \rho_{g,\sigma}(t), \quad (5.33)$$

$$\varepsilon_g(\mathbf{y}_{\sigma}, t) \approx \varepsilon_{g,\sigma}(t), \quad (5.34)$$

$$\mathbf{u}_g(\mathbf{y}_{\sigma}, t) \approx \mathbf{u}_{g,\sigma}(t). \quad (5.35)$$

In the cell-centered finite volume scheme, (5.30)–(5.32) are the primary quantities, i.e., the quantities directly stored in memory. The quantities (5.33)–(5.35) evaluated at the face¹ must be interpolated using the primary quantities. The interpolation is achieved by the Godunov upwind scheme [30]. The upwind formula for $\sigma = K|L \in \mathcal{E}_{\text{int}}$ reads:

$$\text{upwind}_{\sigma}(U(t), \mathbf{u}(t)) = \begin{cases} U_K(t) & \bar{\mathbf{u}}_{\sigma}(t) \cdot \mathbf{n}_{\sigma} \geq 0, \\ U_L(t) & \bar{\mathbf{u}}_{\sigma}(t) \cdot \mathbf{n}_{\sigma} < 0, \end{cases} \quad (5.36)$$

where $U : \mathcal{T} \rightarrow \mathbb{R}^s$, $s \in \{1, d\}$, is an arbitrary quantity mapped to the mesh and $\bar{\mathbf{u}}_{\sigma}$ is the linear interpolation of the velocity corresponding to the same material as the quantity U at σ . For example, the corresponding velocities to the quantities ε_g , ε_s are \mathbf{u}_g , \mathbf{u}_s respectively. The linear interpolation of the velocity is calculated as

$$\bar{\mathbf{u}}_{i,\sigma}(t) = \begin{cases} \alpha \mathbf{u}_{i,K}(t) + (1 - \alpha) \mathbf{u}_{i,L}(t) & \forall \sigma = K | L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma = K | L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \mathbf{u}_{i,\text{in}}(t) & \forall \sigma = K | L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \mathbf{u}_{i,K}(t) & \forall \sigma = K | L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases} \quad (5.37)$$

where $i \in \{g, s\}$ and $\alpha \in [0, 1]$ satisfies $\alpha \mathbf{x}_K + (1 - \alpha) \mathbf{x}_L = \mathbf{y}_{\sigma}$.

¹ $\sigma \in \mathcal{E}$ represents a mesh edge in 2D and a mesh face in 3D. In this chapter, σ will be called a "face" irrespective of the problem dimension d .

The final upwind scheme for approximating the quantities at edges reads

$$\rho_g(\mathbf{y}_\sigma, t) \approx \rho_{g,\sigma}(t) = \begin{cases} \text{upwind}_\sigma(\rho_g(t), \mathbf{u}_g(t)) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \rho_{g,K}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap (\Gamma_{\text{wall}} \cup \Gamma_{\text{in}}), \\ \frac{p_{g,\text{out}}}{R_{\text{spec}}T} & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases} \quad (5.38)$$

$$\varepsilon_g(\mathbf{y}_\sigma, t) \approx \varepsilon_{g,\sigma}(t) = \begin{cases} \text{upwind}_\sigma(\varepsilon_g(t), \mathbf{u}_g(t)) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \varepsilon_{g,K}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ 1 - \varepsilon_{s,\sigma,\text{in}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \varepsilon_{g,\sigma,\text{out}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases} \quad (5.39)$$

$$\mathbf{u}_g(\mathbf{y}_\sigma, t) \approx \mathbf{u}_{g,\sigma}(t) = \begin{cases} \text{upwind}_\sigma(\mathbf{u}_g(t), \mathbf{u}_g(t)) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \mathbf{u}_{g,\text{in}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \mathbf{u}_{g,K}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases} \quad (5.40)$$

For the sake of readability, the dependence of the approximations on time is omitted. Finally, we arrive at the approximation

$$\int_K (\rho_g \varepsilon_g) d\mathbf{x} \approx \rho_{g,K} \varepsilon_{g,K} m(K), \quad (5.41)$$

$$\int_\sigma \rho_g \varepsilon_g (\mathbf{u}_g \cdot \mathbf{n}) dS \approx \rho_{g,\sigma} \varepsilon_{g,\sigma} (\bar{\mathbf{u}}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma), \quad (5.42)$$

where $\mathbf{n}_{K,\sigma}$ is the normal vector to the face σ pointing in the outward direction.

To approximate the diffusion term, we use central differentiation with respect to K and get

$$\int_\sigma \frac{\partial(\rho_g \varepsilon_g)}{\partial \mathbf{n}} dS \approx D_{1,\sigma} = \begin{cases} \frac{\rho_{g,L} \varepsilon_{g,L} - \rho_{g,K} \varepsilon_{g,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} m(\sigma) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \frac{\rho_{g,K} \varepsilon_{g,\text{in}} - \rho_{g,K} \varepsilon_{g,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} m(\sigma) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ 0 & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases}, \quad (5.43)$$

The final form of the approximation of equation (5.26) is

$$\frac{d}{dt} (\rho_{g,K} \varepsilon_{g,K}) = \frac{1}{m(K)} \left(- \sum_{\sigma \in \mathcal{E}_K} \rho_{g,\sigma} \varepsilon_{g,\sigma} (\bar{\mathbf{u}}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma) + \lambda_g \sum_{\sigma \in \mathcal{E}_K} D_{1,\sigma} \right). \quad (5.44)$$

Conservation of Momentum of the Gaseous Phase

A similar procedure leads to the spatial discretization of the second equation of (5.24). First, we integrate over a control volume $K \in \mathcal{T}$ and get

$$\begin{aligned} \int_K \frac{\partial}{\partial t} (\rho_g \varepsilon_g \mathbf{u}_g) d\mathbf{x} &= - \int_K \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g \otimes \mathbf{u}_g) d\mathbf{x} - \int_K \varepsilon_g \nabla p_g d\mathbf{x} + \int_K \nabla \cdot (\varepsilon_g \mathbb{T}_g) d\mathbf{x} \\ &+ \int_K \lambda_g \Delta (\rho_g \varepsilon_g \mathbf{u}_g) d\mathbf{x} + \int_K [\rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g)] d\mathbf{x}. \end{aligned} \quad (5.45)$$

The integration of a vector is defined component-wise. Then, the Green-Gauss theorem is applied to obtain

$$\int_K \nabla \cdot (\rho_g \varepsilon_g \mathbf{u}_g \otimes \mathbf{u}_g) \, d\mathbf{x} = \oint_{\partial K} \rho_g \varepsilon_g \mathbf{u}_g (\mathbf{u}_g \cdot \mathbf{n}) \, dS, \quad (5.46)$$

$$\int_K \nabla \cdot (\varepsilon_g \mathbb{T}_g) \, d\mathbf{x} = \int_K \varepsilon_g \mathbb{T}_g \cdot \mathbf{n} \, dS, \quad (5.47)$$

$$\int_K \lambda_g \Delta (\rho_g \varepsilon_g \mathbf{u}_g) \, d\mathbf{x} = \oint_{\partial K} \lambda_g \frac{\partial (\rho_g \varepsilon_g \mathbf{u}_g)}{\partial \mathbf{x}} \cdot \mathbf{n} \, dS = \oint_{\partial K} \lambda_g \frac{\partial (\rho_g \varepsilon_g \mathbf{u}_g)}{\partial \mathbf{n}} \, dS, \quad (5.48)$$

where $(\mathbb{T}_g \cdot \mathbf{n})_i = \tau_{g,i} \cdot \mathbf{n}$, $i \in \{1, 2, \dots, d\}$. Consecutively, the equation (5.45) after application of the Leibniz integral rule to the differentiation with respect to time assumes the form

$$\begin{aligned} \frac{d}{dt} \int_K (\rho_g \varepsilon_g \mathbf{u}_g) \, d\mathbf{x} &= - \oint_{\partial K} \rho_g \varepsilon_g \mathbf{u}_g (\mathbf{u}_g \cdot \mathbf{n}) \, dS - \int_K \varepsilon_g \nabla p_g \, d\mathbf{x} + \oint_{\partial K} \varepsilon_g \mathbb{T}_g \cdot \mathbf{n} \, dS \\ &+ \oint_{\partial K} \lambda_g \frac{\partial (\rho_g \varepsilon_g \mathbf{u}_g)}{\partial \mathbf{n}} \, dS + \int_K [\rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g)] \, d\mathbf{x}. \end{aligned} \quad (5.49)$$

Using the mesh property $\partial K = \cup_{\sigma \in \mathcal{E}_K} \sigma$ one finds that

$$\begin{aligned} \frac{d}{dt} \int_K (\rho_g \varepsilon_g \mathbf{u}_g) \, d\mathbf{x} &= - \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \rho_g \varepsilon_g \mathbf{u}_g (\mathbf{u}_g \cdot \mathbf{n}_{K,\sigma}) \, dS - \int_K \varepsilon_g \nabla p_g \, d\mathbf{x} \\ &+ \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \varepsilon_g \mathbb{T}_g \cdot \mathbf{n}_{K,\sigma} \, dS + \sum_{\sigma \in \mathcal{E}_K} \lambda_g \int_{\sigma} \frac{\partial (\rho_g \varepsilon_g \mathbf{u}_g)}{\partial \mathbf{n}_{K,\sigma}} \, dS \\ &+ \int_K [\rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g)] \, d\mathbf{x}. \end{aligned} \quad (5.50)$$

Using the relations (5.30)–(5.35), the integrals in (5.50) are approximated as

$$\int_K (\rho_g \varepsilon_g \mathbf{u}_g) \, d\mathbf{x} \approx \rho_{g,K} \varepsilon_{g,K} \mathbf{u}_{g,K} m(K), \quad (5.51)$$

$$\int_{\sigma} \rho_g \varepsilon_g \mathbf{u}_g (\mathbf{u}_g \cdot \mathbf{n}_{K,\sigma}) \, dS \approx \rho_{g,\sigma} \varepsilon_{g,\sigma} \mathbf{u}_{g,\sigma} (\mathbf{u}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma), \quad (5.52)$$

$$\int_K \varepsilon_g \nabla p_g \, d\mathbf{x} \approx \varepsilon_{g,K} \nabla p_{g,K} m(K), \quad (5.53)$$

$$\int_{\sigma} \varepsilon_g \mathbb{T}_g \cdot \mathbf{n}_{K,\sigma} \, dS \approx \bar{\varepsilon}_{g,\sigma} \mathbb{T}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma}, \quad (5.54)$$

$$\int_K [\rho_g \mathbf{g} + \beta_{gs} (\mathbf{u}_s - \mathbf{u}_g)] \, d\mathbf{x} \approx \rho_{g,K} \mathbf{g} + \beta_{gs,K} (\mathbf{u}_{s,K} - \mathbf{u}_{g,K}), \quad (5.55)$$

where $\beta_{gs,K}$ is obtained by replacing (5.8)–(5.10) all unknown quantities in by their respective approximations at \mathbf{x}_K , i.e., the relations (5.30)–(5.32) and (5.71), (5.72) are used. The term $\bar{\varepsilon}_{g,\sigma}$ is calculated as

$$\bar{\varepsilon}_{g,\sigma}(t) = \begin{cases} \alpha \varepsilon_{g,\sigma}(t) + (1 - \alpha) \varepsilon_{g,\sigma}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \varepsilon_{g,K}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ 1 - \varepsilon_{s,\sigma,\text{in}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \varepsilon_{g,\sigma,\text{out}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases} \quad (5.56)$$

where $\alpha \in [0, 1]$ satisfies $\alpha \mathbf{x}_K + (1 - \alpha) \mathbf{x}_L = \mathbf{y}_{\sigma}$. The quantities $\varepsilon_{g,\sigma}$, $\rho_{g,\sigma}$, $\mathbf{u}_{g,\sigma}$ are defined according to equations (5.39), (5.38), (5.37). Consecutively, the pressure gradient is approximated using the Green-Gauss theorem

$$\int_K \nabla p_g \, d\mathbf{x} = \int_{\partial K} p_g \mathbf{n} \, dS \approx \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma), \quad (5.57)$$

where the symbol $\bar{p}_{g,\sigma}$, in (5.57) is defined as

$$\bar{p}_{g,\sigma}(t) = \begin{cases} \alpha p_{g,K}(t) + \beta p_{g,L}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ R_{\text{spec}} \rho_K(t) T & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap (\Gamma_{\text{wall}} \cup \Gamma_{\text{in}}) \\ p_{\text{atm}} & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases} \quad (5.58)$$

At the same time, the left hand side integral expression can be approximated by the rectangle rule as

$$\int_K \nabla p_g d\mathbf{x} \approx \nabla p_{g,K} m(K). \quad (5.59)$$

After combining equations (5.57), (5.59), we arrive at the final approximation

$$\nabla p_{g,K} \approx \frac{1}{m(K)} \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma), \quad (5.60)$$

The final approximation of (5.53) reads

$$\int_K \varepsilon_g \nabla p_g d\mathbf{x} \approx \varepsilon_{g,K} \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma).$$

The diffusion term is approximated similarly to (5.43):

$$\int_{\sigma} \frac{\partial(\rho_g \varepsilon_g \mathbf{u}_g)}{\partial \mathbf{n}_{K,\sigma}} dS \approx D_{2,\sigma} = \begin{cases} \frac{\rho_{g,L} \varepsilon_{g,L} \mathbf{u}_{g,L} - \rho_{g,K} \varepsilon_{g,K} \mathbf{u}_{g,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} m(\sigma) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \frac{\rho_{g,K} \varepsilon_{g,\text{in}} \mathbf{u}_{g,\text{in}} - \rho_{g,K} \varepsilon_{g,K} \mathbf{u}_{g,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} m(\sigma) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases} \quad (5.61)$$

The symbol $\mathbb{T}_{g,\sigma}$ represents an approximation of $\mathbb{T}_g(\mathbf{y}_\sigma, t)$ defined according to (5.4). The calculation of the stress tensor requires to calculate the partial derivatives of \mathbf{u}_g . However, especially in the 3D case, there is no convenient method to calculate the approximation of the derivative of velocity directly. The reason is that the mesh provides the face normals only, the tangential vectors are not provided. Therefore, to calculate

$$\nabla \mathbf{u}_g \equiv \left(\frac{\partial u_{g,j}}{\partial x_i} \right)_{i,j=1}^d, \quad (5.62)$$

we apply the Gauss gradient scheme [30] which utilizes Gauss-Green theorem:

$$\int_K \nabla \mathbf{u}_g d\mathbf{x} = \int_{\partial K} \mathbf{u}_g \otimes \mathbf{n} dS \approx \sum_{\sigma \in \mathcal{E}_K} \bar{\mathbf{u}}_{g,\sigma} \otimes \mathbf{n}_\sigma m(\sigma), \quad (5.63)$$

where $\bar{\mathbf{u}}_{g,\sigma}$ is given by (5.37). We define the integration of a matrix element-wise. After approximation of the left hand side of (5.63)

$$\int_K \nabla \mathbf{u}_g d\mathbf{x} \approx \nabla \mathbf{u}_{g,K} m(K), \quad (5.64)$$

we arrive at the final approximation

$$\nabla \mathbf{u}_{g,K} = \frac{1}{m(K)} \sum_{\sigma \in \mathcal{E}_K} \bar{\mathbf{u}}_{g,\sigma} \otimes \mathbf{n}_{\sigma,K} m(\sigma). \quad (5.65)$$

Afterward, the approximation of $\nabla \mathbf{u}_g$ at \mathbf{y}_σ is calculated as the linear interpolation

$$\nabla \mathbf{u}_{g,\sigma} = \begin{cases} \alpha \nabla \mathbf{u}_K + (1 - \alpha) \nabla \mathbf{u}_L & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \nabla \mathbf{u}_{g,K} & \forall \sigma \in \mathcal{E}_{\text{ext}}. \end{cases}, \quad (5.66)$$

where $\alpha \in [0, 1]$ satisfies $\alpha \mathbf{x}_K + (1 - \alpha) \mathbf{x}_L = \mathbf{y}_\sigma$. The approximation of the stress tensor evaluated at the face σ has the form:

$$\mathbb{T}_{g,\sigma} = \mu \left(\nabla \mathbf{u}_{g,\sigma} + (\nabla \mathbf{u}_{g,\sigma})^T \right) - \nabla \cdot \mathbf{u}_{g,\sigma} \mathbb{I}. \quad (5.67)$$

Finally, the application of FVM to the second equation of (5.24) results in

$$\begin{aligned} \frac{d}{dt} (\rho_{g,K} \varepsilon_{g,K} \mathbf{u}_{g,K}) &= \frac{1}{m(K)} \left(- \sum_{\sigma \in \mathcal{E}_K} \rho_{g,\sigma} \varepsilon_{g,\sigma} \mathbf{u}_{g,\sigma} (\bar{\mathbf{u}}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma) + \varepsilon_g \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma) \right. \\ &\quad \left. + \sum_{\sigma \in \mathcal{E}_K} [\bar{\varepsilon}_{g,\sigma} \mathbb{T}_{g,\sigma} \cdot \mathbf{n}_{K,\sigma} + \lambda_g D_{2,\sigma}] m(\sigma) \right) \\ &\quad + \rho_{g,K} \mathbf{g} + \beta_{gs,K} (\mathbf{u}_{s,K} - \mathbf{u}_{g,K}). \end{aligned} \quad (5.68)$$

Conservation of Mass of the Solid Phase

The procedure of applying the FVM scheme on the mass conservation law for the solid state is exactly the same as in Section 5.2.2. We integrate the third row of equation (5.24) over a control volume $K \in \mathcal{T}$ and arrive at

$$\int_K \frac{\partial}{\partial t} (\rho_s \varepsilon_s) d\mathbf{x} = \int_K (-\rho_s \varepsilon_s \nabla \cdot \mathbf{u}_s + \lambda_s \Delta (\rho_s \varepsilon_s)) d\mathbf{x}. \quad (5.69)$$

Next, we use the Green-Gauss theorem and Leibniz integral rule:

$$\begin{aligned} \frac{d}{dt} \int_K (\rho_s \varepsilon_s) d\mathbf{x} &= - \oint_{\partial K} \rho_s \varepsilon_s (\mathbf{u}_s \cdot \mathbf{n}) dS + \oint_{\partial K} \lambda_s \frac{\partial (\rho_s \varepsilon_s)}{\partial \mathbf{n}} dS, \\ &= - \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \rho_s \varepsilon_s (\mathbf{u}_s \cdot \mathbf{n}) dS + \sum_{\sigma \in \mathcal{E}_K} \lambda_s \int_{\sigma} \frac{\partial (\rho_s \varepsilon_s)}{\partial \mathbf{n}} dS. \end{aligned} \quad (5.70)$$

We approximate the quantities similarly to relations (5.30)–(5.35):

$$\varepsilon_s(\mathbf{x}_K, t) \approx \varepsilon_{s,K}(t), \quad (5.71)$$

$$\mathbf{u}_s(\mathbf{x}_K, t) \approx \mathbf{u}_{s,K}(t), \quad (5.72)$$

$$\varepsilon_s(\mathbf{y}_\sigma, t) \approx \varepsilon_{s,\sigma}(t), \quad (5.73)$$

$$\mathbf{u}_s(\mathbf{y}_\sigma, t) \approx \mathbf{u}_{s,\sigma}(t). \quad (5.74)$$

Again the quantities (5.71)–(5.72) are primary and the quantities (5.73)–(5.74) evaluated on the faces have to be interpolated as

$$\varepsilon_{s,\sigma}(t) = \begin{cases} \text{upwind}_\sigma(\varepsilon_s(t), \mathbf{u}_s(t)) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \varepsilon_{s,K}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \varepsilon_{s,\sigma,\text{in}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \varepsilon_{s,\sigma,\text{out}}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases} \quad (5.75)$$

$$\mathbf{u}_{s,\sigma}(t) = \begin{cases} \text{upwind}_\sigma(\mathbf{u}_s(t), \mathbf{u}_s(t)) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \mathbf{u}_{s,\text{in}}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \mathbf{u}_{s,K}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases} \quad (5.76)$$

For the sake of readability, the dependency on time is omitted. Finally, we arrive at the approximations

$$\int_K (\rho_s \varepsilon_s) d\mathbf{x} \approx \rho_s \varepsilon_{s,K} m(K), \quad (5.77)$$

$$\int_\sigma \rho_s \varepsilon_s (\mathbf{u}_s \cdot \mathbf{n}) dS \approx \rho_s \varepsilon_{s,\sigma} (\mathbf{u}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma), \quad (5.78)$$

where ρ_s is constant, see Table 5.1. The approximation of the diffusion term is realized as

$$\int_\sigma \frac{\partial(\rho_s \varepsilon_s)}{\partial \mathbf{n}} dS \approx D_{3,\sigma} = \begin{cases} \frac{\rho_s \varepsilon_{s,L} - \rho_s \varepsilon_{s,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \frac{\rho_s \varepsilon_{s,\text{in}} - \rho_s \varepsilon_{s,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases} \quad (5.79)$$

Finally, the obtained semi-discrete scheme reads

$$\frac{d}{dt} (\rho_s \varepsilon_{s,K}) = \frac{1}{m(K)} \left(- \sum_{\sigma \in \mathcal{E}_K} \rho_s \varepsilon_{s,\sigma} (\bar{\mathbf{u}}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma) + \lambda_s \sum_{\sigma \in \mathcal{E}_K} D_{3,\sigma} \right). \quad (5.80)$$

Conservation of Momentum of the Solid Phase

The spatial discretization of the fourth equation is almost the same as in Section 5.2.2. We integrate the last row of equation (5.24) over a control volume $K \in \mathcal{T}$ to obtain

$$\begin{aligned} \int_K \frac{\partial}{\partial t} (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} &= - \int_K \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s \otimes \mathbf{u}_s) d\mathbf{x} - \int_K G(\varepsilon_g) \nabla \varepsilon_s d\mathbf{x} \\ &\quad - \int_K \varepsilon_s \nabla p_g d\mathbf{x} + \int_K \nabla \cdot (\varepsilon_s \mathbb{T}_s) d\mathbf{x} + \int_K \lambda_s \Delta (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} \\ &\quad + \int_K [(\rho_s - \rho_g) \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s)] d\mathbf{x}. \end{aligned} \quad (5.81)$$

Application of the Green-Gauss theorem on the terms on the right hand side of equation (5.81) yields

$$\int_K \nabla \cdot (\rho_s \varepsilon_s \mathbf{u}_s \otimes \mathbf{u}_s) d\mathbf{x} = \oint_{\partial K} \rho_s \varepsilon_s \mathbf{u}_s (\mathbf{u}_s \cdot \mathbf{n}) dS, \quad (5.82)$$

$$\int_K \nabla \cdot (\varepsilon_s \mathbb{T}_s) d\mathbf{x} = \int_{\partial K} \varepsilon_s \mathbb{T}_s \cdot \mathbf{n} dS, \quad (5.83)$$

$$\int_K \lambda_s \Delta (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} = \oint_{\partial K} \lambda_s \frac{\partial(\rho_s \varepsilon_s \mathbf{u}_s)}{\partial \mathbf{x}} \cdot \mathbf{n} dS = \oint_{\partial K} \lambda_g \frac{\partial(\rho_s \varepsilon_s \mathbf{u}_s)}{\partial \mathbf{n}} dS, \quad (5.84)$$

where $(\mathbb{T}_s \cdot \mathbf{n})_i = \tau_{s,i} \cdot \mathbf{n}$. Consecutively, after application of the Leibniz integral rule on (5.81), we arrive at

$$\begin{aligned} \frac{d}{dt} \int_K (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} &= - \oint_{\partial K} \rho_s \varepsilon_s \mathbf{u}_s (\mathbf{u}_s \cdot \mathbf{n}) dS - \int_K G(\varepsilon_g) \nabla \varepsilon_s d\mathbf{x} \\ &\quad - \int_K \varepsilon_s \nabla p_g d\mathbf{x} + \oint_{\partial K} \varepsilon_s \mathbb{T}_s \cdot \mathbf{n} dS + \oint_{\partial K} \lambda_s \frac{\partial(\rho_s \varepsilon_s \mathbf{u}_s)}{\partial \mathbf{n}} dS \\ &\quad + \int_K [(\rho_s - \rho_g) \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s)] d\mathbf{x}. \end{aligned} \quad (5.85)$$

Integrals over ∂K can be split as

$$\begin{aligned} \frac{d}{dt} \int_K (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} &= - \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \rho_s \varepsilon_s \mathbf{u}_s (\mathbf{u}_s \cdot \mathbf{n}_{K,\sigma}) dS - \int_K G(\varepsilon_g) \nabla \varepsilon_s d\mathbf{x} \\ &\quad - \int_K \varepsilon_s \nabla p_g d\mathbf{x} + \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \varepsilon_s \mathbb{T}_s \cdot \mathbf{n}_{K,\sigma} dS + \sum_{\sigma \in \mathcal{E}_K} \lambda_s \int_{\sigma} \frac{\partial (\rho_s \varepsilon_s \mathbf{u}_s)}{\partial \mathbf{n}_{K,\sigma}} dS \\ &\quad + \int_K [(\rho_s - \rho_g) \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s)] d\mathbf{x}. \end{aligned} \quad (5.86)$$

Based on the approximations (5.71)–(5.74), the integrals are approximated in the same manner as in Section 5.2.2:

$$\int_K (\rho_s \varepsilon_s \mathbf{u}_s) d\mathbf{x} \approx \rho_{s,K} \varepsilon_{s,K} \mathbf{u}_{s,K} m(K), \quad (5.87)$$

$$\int_{\sigma} \rho_s \varepsilon_s \mathbf{u}_s (\mathbf{u}_s \cdot \mathbf{n}_{K,\sigma}) dS \approx \rho_{s,\sigma} \varepsilon_{s,\sigma} \mathbf{u}_{s,\sigma} (\mathbf{u}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma}) m(\sigma), \quad (5.88)$$

$$\int_K G(\varepsilon_g) \nabla \varepsilon_s d\mathbf{x} \approx G(\varepsilon_{g,\sigma}) \sum_{\sigma \in \mathcal{E}_K} \bar{\varepsilon}_{s,\sigma} \mathbf{n}_{K,\sigma} m(\sigma), \quad (5.89)$$

$$\int_K \varepsilon_s \nabla p_g d\mathbf{x} \approx \varepsilon_s \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma), \quad (5.90)$$

$$\int_{\sigma} \varepsilon_s \mathbb{T}_s \cdot \mathbf{n}_{K,\sigma} dS \approx \bar{\varepsilon}_{s,\sigma} \mathbb{T}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma}, \quad (5.91)$$

$$\int_K [\rho_s \mathbf{g} + \beta_{gs} (\mathbf{u}_g - \mathbf{u}_s)] d\mathbf{x} \approx \rho_{s,K} \mathbf{g} + \beta_{gs,K} (\mathbf{u}_{g,K} - \mathbf{u}_{s,K}). \quad (5.92)$$

The quantity $\bar{\varepsilon}_{s,\sigma}$ is calculated as

$$\bar{\varepsilon}_{s,\sigma}(t) = \begin{cases} \alpha \varepsilon_{g,\sigma}(t) + (1 - \alpha) \varepsilon_{g,\sigma}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \varepsilon_{s,K}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \varepsilon_{s,\sigma,\text{in}}(t) & \forall \sigma = K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ \varepsilon_{s,\sigma,\text{out}}(t) & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}, \end{cases}$$

where $\alpha \in [0, 1]$ satisfies $\alpha \mathbf{x}_K + (1 - \alpha) \mathbf{x}_L = \mathbf{y}_{\sigma}$. The quantities $\varepsilon_{s,\sigma}$, $\mathbf{u}_{s,\sigma}$ are defined according to equations (5.75), (5.76). The approximation of the diffusion term reads

$$\int_{\sigma} \frac{\partial (\rho_s \varepsilon_s \mathbf{u}_s)}{\partial \mathbf{n}_{K,\sigma}} dS \approx D_{4,\sigma} = \begin{cases} \frac{\rho_s \varepsilon_{s,L} \mathbf{u}_{s,L} - \rho_s \varepsilon_{s,K} \mathbf{u}_{s,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{int}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{wall}}, \\ \frac{\rho_s \varepsilon_{s,\text{in}} \mathbf{u}_{s,\text{in}} - \rho_s \varepsilon_{s,K} \mathbf{u}_{s,K}}{m(\mathcal{D}_{\mathbf{x}_K, \mathbf{x}_L})} & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{in}}, \\ 0 & \forall \sigma \in K \mid L \in \mathcal{E}_{\text{ext}} \cap \Gamma_{\text{out}}. \end{cases}, \quad (5.93)$$

The symbol $\mathbb{T}_{s,\sigma}$ denotes an approximation of $\mathbb{T}_s(\mathbf{y}_{\sigma}, t)$ defined according to (5.4). The calculation of the stress tensor requires to calculate the gradient of \mathbf{u}_s .

Similarly to the case of the gaseous phase, we utilize the Gauss gradient scheme and arrive at the approximation

$$\nabla \mathbf{u}_{s,K} = \frac{1}{m(K)} \sum_{\sigma \in \mathcal{E}_K} \bar{\mathbf{u}}_{s,\sigma} \otimes \mathbf{n}_{\sigma,K} m(\sigma), \quad (5.94)$$

similarly to the equation (5.65). The approximation of $\nabla \mathbf{u}_s$ at \mathbf{y}_{σ} is calculated as

$$\nabla \mathbf{u}_{s,\sigma} = \begin{cases} \alpha \nabla \mathbf{u}_{s,K} + (1 - \alpha) \nabla \mathbf{u}_{s,L} & \forall \sigma = K \mid L \in \mathcal{E}_{\text{int}}, \\ \nabla \mathbf{u}_{s,K} & \forall \sigma \in \mathcal{E}_{\text{ext}}. \end{cases}, \quad (5.95)$$

where $\alpha \in [0, 1]$ satisfies $\alpha \mathbf{x}_K + \beta \mathbf{x}_L = \mathbf{y}_\sigma$. The approximation of the stress tensor evaluated at the face σ has the form:

$$\mathbb{T}_{s,\sigma} = \mu \left(\nabla \mathbf{u}_{s,\sigma} + (\nabla \mathbf{u}_{s,\sigma})^T \right) - \nabla \cdot \mathbf{u}_{s,\sigma} \mathbb{I}. \quad (5.96)$$

Finally, the application of FVM to the fourth equation of relation (5.24) results in:

$$\begin{aligned} \frac{d}{dt} (\rho_s \varepsilon_{s,K} \mathbf{u}_{s,K}) &= \frac{1}{m(K)} \left(- \sum_{\sigma \in \mathcal{E}_K} [\rho_s \varepsilon_{s,\sigma} \mathbf{u}_{s,\sigma} (\bar{\mathbf{u}}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma})] m(\sigma) \right. \\ &\quad - G(\varepsilon_{g,\sigma}) \sum_{\sigma \in \mathcal{E}_K} \bar{\varepsilon}_{s,\sigma} \mathbf{n}_{K,\sigma} m(\sigma) - \varepsilon_s \sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma) \\ &\quad \left. + \sum_{\sigma \in \mathcal{E}_K} [\bar{\varepsilon}_{s,\sigma} \mathbb{T}_{s,\sigma} \cdot \mathbf{n}_{K,\sigma} + \lambda_s D_{4,\sigma}] m(\sigma) \right) \\ &\quad + (\rho_{s,K} - \rho_{g,K}) \mathbf{g} + \beta_{g_s,K} (\mathbf{u}_{g,K} - \mathbf{u}_{s,K}). \end{aligned} \quad (5.97)$$

5.2.3 Treatment of Non-planar Faces

In Section 2.7, the treatment of non-planar faces is discussed in terms of calculation of the mesh properties. This section presents the effect of non-planar faces on the computation. According to the discussion in Section 2.7, the computation over faces is to be split according to the tessellation. Thanks to the fact that all the faces created by the tessellation have the same neighboring cells, the calculation can be modified to calculate the sum of the requested quantities at once.

First, we denote the tessellation of a face $\sigma = K|L \in \mathcal{E}_{\text{in}}$

$$\mathcal{T}(\sigma) = \bigcup_{e \in N^1(\sigma)} \Delta_e, \quad (5.98)$$

where Δ_e the triangle defined in Section 2.7. In order to simplify the notation, the faces Δ_e created as tessellation of face σ will be denoted as σ' . The effect of the tessellation of faces on the equations will be demonstrated on the term

$$\sum_{\sigma \in \mathcal{E}_K} \bar{p}_{g,\sigma} \mathbf{n}_{K,\sigma} m(\sigma), \quad (5.99)$$

from (5.97). Focusing on one face σ , the tessellation reads

$$\begin{aligned} \sum_{\sigma' \in \mathcal{T}(\sigma)} \bar{p}_{g,\sigma'} \mathbf{n}_{K,\sigma'} m(\sigma) &\stackrel{(5.58)}{=} \sum_{\sigma' \in \mathcal{T}(\sigma)} (\alpha_{\sigma'} p_{g,K} + \beta_{\sigma'} p_{g,L}) \mathbf{n}_{K,\sigma'} m(\sigma') \\ &= \underbrace{p_{g,K} \sum_{\sigma' \in \mathcal{T}(\sigma)} \alpha_{\sigma'} \mathbf{n}_{K,\sigma'} m(\sigma')} + \underbrace{p_{g,L} \sum_{\sigma' \in \mathcal{T}(\sigma)} \beta_{\sigma'} \mathbf{n}_{K,\sigma'} m(\sigma')}, \end{aligned} \quad (5.100)$$

where the coefficients of linear interpolation read

$$\alpha_{\sigma'} = \frac{|\mathbf{x}_K - \mathbf{y}_{\sigma'}|}{|\mathbf{x}_K - \mathbf{y}_{\sigma'}| + |\mathbf{x}_L - \mathbf{y}_{\sigma'}|}, \quad (5.101)$$

$$\beta_{\sigma'} = \frac{|\mathbf{x}_L - \mathbf{y}_{\sigma'}|}{|\mathbf{x}_K - \mathbf{y}_{\sigma'}| + |\mathbf{x}_L - \mathbf{y}_{\sigma'}|}, \quad (5.102)$$

where $\mathbf{y}_{\sigma'} \in \sigma'$. The underlined terms in (5.100) can be calculated in advance as an additional property of a mesh with non-planar faces. Using these terms, the computation over one face can be calculated without an additional loop.

In the presented example of tessellation of term (5.99), the calculation can be shortened without any further assumptions. However, in processing of the terms containing quantities calculated by upwind (5.36), the same approach can be used only if the orientation of velocity is the same with respect to all faces in the tessellation of the face σ , i.e.

$$(\forall \sigma' \in \mathcal{T}(\sigma)) (\bar{\mathbf{u}}_{\sigma'} \cdot \mathbf{n}_{\sigma',K} \geq 0) \vee (\forall \sigma' \in \mathcal{T}(\sigma)) (\bar{\mathbf{u}}_{\sigma'} \cdot \mathbf{n}_{\sigma',K} < 0).$$

In addition, it is possible to choose $\mathbf{y}_{\sigma'} = \mathbf{x}_{\sigma}^* \forall \sigma' \in \mathcal{T}(\sigma)$ where \mathbf{x}_{σ}^* is defined in (2.20). The treatment of non-planar faces in finite volume schemes for selected problems is discussed, e.g., in [37].

5.2.4 Temporal Discretization

Denote by \mathbf{w} the vector of values of the approximate solution of (5.6) in all cells of the mesh at time t , i.e.

$$\mathbf{w}(t) = \begin{pmatrix} \rho_{g,K}(t) \varepsilon_{g,K}(t) \quad \forall K \in \mathcal{T} \\ \rho_{g,K}(t) \varepsilon_{g,K}(t) \mathbf{u}_{g,K}(t) \quad \forall K \in \mathcal{T} \\ \rho_s \varepsilon_{s,K}(t) \quad \forall K \in \mathcal{T} \\ \rho_s \varepsilon_{s,K}(t) \mathbf{u}_{s,K}(t) \quad \forall K \in \mathcal{T} \end{pmatrix}. \quad (5.103)$$

Then the equations (5.44), (5.68), (5.80), (5.97) of the semi-discrete scheme together represent a system of ODEs in the form:

$$\frac{d\mathbf{w}}{dt} = \mathbf{f}(t, \mathbf{w}) \quad (5.104)$$

In this work, we employ the Merson version of Runge-Kutta explicit scheme for numerical solution of (5.104). The scheme is described in Section 4.3 and employs an adaptive time stepping algorithm to ensure stability.

5.3 Realization of the Computation in the GTMesh framework

In this section, the implementation of the numerical algorithm with emphasis on the utilization of the developed tools is briefly presented. The computation is provided by the `MultiphaseFlow` class template. During the implementation of this generic numerical scheme, the results were compared with the ones from the 2D implementation of the same problem in previous work [40].

The `MultiphaseFlow` class template has the template parameter named `Dim` specifying the dimension of the problem (2 or 3). This class has the interface required by the `RKMSolver` function (see Section 4.3), i.e., the `MeshType` and `ResultType` aliases and the `calculateRHS` member function (see Code listing 5.1). The `calculateRHS` member function calculates the right hand side of (5.104). It accepts `time` and two computational data containers where the second is used for the output. In order to introduce the input and output `MeshDataContainer` structures of `calculateRHS`, we have to describe the data structure representing the quantities of two-phase flow.

The structure representing the two-phase flow quantities (e.g., densities, velocities and volume fractions of the phases) is named `FlowData`. The description of `FlowData` is in Code listing 5.2. In the `MultiphaseFlow` structure, `FlowData` is mapped to the cells of the mesh using `MeshDataContainer`. The `FlowData` structure has both the default arithmetic traits and the default IO traits defined.

The default arithmetic traits of `FlowData` contain the primary quantities (see (5.103)). Therefore, `MeshDataContainer` of `FlowData` can be directly processed by the `RKMSolver` function. Since we do not want the primary quantities to be exported, the default IO traits contain the dependent quantities, e.g., density and velocity. See the definition of default traits in Code listing 5.2.

Code listing 5.1 The definition of the `MultiphaseFlow` class template. At first, `MultiphaseFlow` provides several aliases as `MeshType` and `ResultType`. Then, the definition of the attributes follows, i.e., `mesh` (instance of `MeshType`), `meshData` (auxiliary computation data mapped to cells and faces) and parameters of the two-phase flow problem. The definitions of the utilized data structures are presented in Code listings 5.2 and 5.3. Next, the definition of the mandatory `calculateRHS` member function follows. This function provides the calculation of the right hand side of (5.104). The `setupMeshData` member function loads the mesh from the given file and prepares the auxiliary mesh quantities in the `meshData` container. The `exportData` member function exports the mesh together with the numerical solution into a VTK file. See the implementation of `setupMeshData` and `exportData` in Code listing 5.5. The writer and reader member attributes are utilized to prevent multiple indexing of the mesh, see Section 2.5. Finally, the computational member functions follows. For an example, see the implementation of calculation of the flux across a face in Code listing 5.4.

```

1  template<unsigned int Dimension, unsigned int ... Reserve>
2  class MultiphaseFlow {
3
4  public: // Compulsory aliases
5      static constexpr unsigned int ProblemDimension = Dimension;
6      using MeshType = UnstructuredMesh< ProblemDimension, size_t, double, Reserve ... >;
7      using ResultType = FlowData< ProblemDimension >;
8
9  public:
10     // Structures containing state data
11
12     MeshType mesh;
13
14     MeshDataContainer< std::tuple< CellData< ProblemDimension >,
15                          FaceData< ProblemDimension > >,
16                          ProblemDimension,
17                          ProblemDimension - 1 > meshData;
18
19     double myu, myu_s;
20     double R_spec;
21     // ** definition of other parameters **
22
23     void calculateRHS( double time,
24                       MeshDataContainer< ResultType, ProblemDimension >& compData,
25                       MeshDataContainer< ResultType, ProblemDimension >& outDeltas);
26
27     // Two different mesh setups for the 2D and the 3D cases
28     template<unsigned int _Dimension = Dimension>
29     typename std::enable_if<_Dimension == 3>::type
30     setupMeshData(const std::string& fileName);
31
32     template<unsigned int _Dimension = Dimension>
33     typename std::enable_if<_Dimension == 2>::type
34     setupMeshData(const std::string& fileName);
35
36     void exportData( double time,
37                     MeshDataContainer<ResultType, ProblemDimension>& compData,
38                     double timeModifier = 1.0);
39 private:
40
41     VTKMeshWriter<ProblemDimension> writer;
42     std::unique_ptr<MeshReader<ProblemDimension>> reader;
43
44     // ** Definitions of computation methods **
45 };

```

Code listing 5.2 The definition of the FlowData structure template. This data structure contains the computational quantities of the two-phase flow problem. Its purpose is to be mapped to the cells of the mesh. Based on the Dimension template parameter, the contained vectors have 2 or 3 coordinates. The structure defines the constant quantities at first, i.e., the specific gas constant (R_spec), temperature (T), density of the solid phase (rho_s). In order to reduce the size of the structure, these attributes are declared as static. Then, the quantities of the two-phase flow are defined, i.e., the density multiplied by the volume fraction of the gaseous phase (rho_g_x_eps_g), momentum of both phases (p_g, p_s), volume fraction of the solid phase (eps_s). We call these quantities primary as the other can be calculated from them. The rest of the functions provide the dependent quantities. Finally, the listing presents the definition of default traits for the FlowData class template.

```

1 // Adds small non-zero constant (1e-7) to prevent zero division.
2 static double reg( double x ) { return x + 1.0e-7; }
3 // Data structure representing the flow quantities in single cell element
4 template< unsigned int Dim >
5 struct FlowData {
6     // ** Constants **
7     // Constants: specific gas constant, Temperature, Density of the solid phase
8     static double R_spec, T, rho_s;
9
10    // ** Primary quantities **
11    // Density multiplied by volume fraction of the gaseous phase
12    double rho_g_x_eps_g;
13    // Volume fraction of solid part of the flow
14    double eps_s;
15    // Momentum of gaseous phase
16    Vector< Dim, double > p_g;
17    // Momentum of solid phase
18    Vector< Dim, double > p_s;
19
20    // ** Dependent quantities **
21    // Volume fraction of gaseous part of the flow
22    double getEps_g() const { return 1.0 - eps_s; }
23    void setEps_g( const double& eps_g ){ eps_s = 1.0 - eps_g; }
24    // Density of the gaseous phase
25    double getRho_g() const { return rho_g_x_eps_g / reg(getEps_g()); }
26    void setRho_g( const double& rho_g ){ rho_g_x_eps_g = getEps_g() * rho_g; }
27    // Velocity of gaseous phase
28    Vector<Dim, double> getVelocityGas() const { return p_g / reg( rho_g_x_eps_g ); }
29    void setVelocityGas( const Vector<Dim, double>& u_g ){ p_g = u_g * rho_g_x_eps_g; }
30    // Pressure of the gaseous phase
31    double getPressure() const { return getRho_g() * R_spec * T; }
32    void setPressure( const double& pressure ){ setRho_g( pressure / ( R_spec * T ) ); }
33    // Velocity of the solid phase
34    Vector<Dim, double> getVelocitySolid() const { return p_s / reg( rho_s * eps_s ); }
35    void setVelocitySolid( const Vector<Dim, double>& u_s ){ p_s = u_s * rho_s * eps_s; }
36 };
37 // Definition of default arithmetic traits for 2D case
38 MAKE_ATTRIBUTE_TRAIT_TEMPLATE_ARITHMETIC(
39     (FlowData<Dim>), (unsigned int Dim), rho_g_x_eps_g, eps_s, p_g, p_s
40 );
41 // Definition of default traits
42 MAKE_CUSTOM_TEMPLATE_TRAIT_IO(
43     (FlowData<Dim>), (unsigned int Dim),
44     "eps_g", std::make_pair(&FlowData<Dim>::getEps_g, &FlowData<Dim>::setEps_g),
45     "pressure", std::make_pair(&FlowData<Dim>::getPressure, &FlowData<Dim>::setPressure),
46     "rho_g", &FlowData<Dim>::rho_g,
47     "eps_s", &FlowData<Dim>::eps_s,
48     "velocity_gas",
49     std::make_pair(&FlowData<Dim>::getVelocityGas, &FlowData<Dim>::setVelocityGas),
50     "velocity_solid",
51     std::make_pair(&FlowData<Dim>::getVelocitySolid, &FlowData<Dim>::setVelocitySolid)
52 );

```

Code listing 5.3 Definition of the structures for auxiliary mesh quantities and computational data. Let us note that the second order tensors were implemented as vectors of vectors. Thanks to the element-wise operations defined, these operations are applicable to `Vector<Dim, Vector<Dim, double>>`. Additionally, this object automatically provides the expected matrix notation interface, i.e., two subscript operators.

```

1  template<unsigned int Dim>
2  struct FaceData {
3      // Measure of the face divided by the distance of the neighboring cell centers
4      double MeasureOverDist;
5      // Measure of hte face element
6      double Measure;
7      // Normal vector of the face
8      Vector<Dim, double> n;
9      // Koeficients of linear combination of the values of the neighboring cells
10     double LeftCellKoeff;
11     double RightCellKoeff
12
13     // Flux of the momentum of the gaseous phase across the face
14     Vector<Dim, double> fluxP_g;
15     // Flux of the momentum of the solid phase across the face
16     Vector<Dim, double> fluxP_s;
17     // Flux of the mass of the solid phase across the face
18     double fluxRho_s;
19     // Flux of the mass of the gaseous phase across the face
20     double fluxRho_g;
21
22     // Gradient of velocity of gas
23     Vector<Dim, Vector<Dim, double>> grad_u_g;
24     // Gradient of velocity of gas
25     Vector<Dim, Vector<Dim, double>> grad_u_s;
26 };

```

```

1  template<unsigned int Dim>
2  struct CellData{
3      // Inverted value of the cell measure
4      double invVolume;
5
6      // Gradients of velocities approximated at cells
7      Vector<Dim, Vector<Dim, double>> grad_u_g;
8      Vector<Dim, Vector<Dim, double>> grad_u_s;
9  };

```

The simulation of two-phase flow requires two additional data structures `CellData` and `FaceData`. The `CellData` structure stores the inverted value of the cell volumes and computed approximations of gradients (5.65) and (5.94). `FaceData` stores, for example, the measure of the face divided by the cells distance, coefficients α and $1 - \alpha$ from (5.37) or the temporary values of fluxes (see Code listing 5.3). These data are mapped to the corresponding elements using `MeshDataContainer`. For better understanding see the definition of attributes of `MultiphaseFlow` in Code listing 5.1.

All the mentioned data structures are templates to be able to specify the dimension (number of components) of the vector or tensor quantities. Thanks to the prepared data structures, the work with the data is very convenient and the code is readable.

Let us focus on the realization of the calculation of (5.104) by the `calculateRHS` function. In equations (5.44), (5.68), (5.80), (5.97) evaluated for adjacent cells K, L , the advective flux across the edge $\sigma = K|L$ has the same absolute value but opposite sign. Thus, flux terms can be evaluated only once using a loop over all the faces of the mesh. Then, the result is added to the left cell and subtracted from the right cell. In order to prevent memory conflicts in parallel computation, we store the result of fluxes over faces in the `FaceData` structure. Then, the fluxes are summed in the next iteration over cells and their boundaries.

Because of the calculation of the gradients of velocities, there must be one additional iteration over faces and cells boundaries. The first iteration calculates the values of the approximation of the velocity gradient in the cells. The second pair of loops calculates the viscous flux. The calculation of the right hand side consists of four cycles:

1. calculate the advective flux of the quantities and tensor product of velocity and normal vector and store them in the `FaceData` structure allocated to the faces,
2. calculate the flux and gradient of velocity in cells by summing the previously calculated quantities over the cell boundary,
3. calculate the viscous fluxes at faces using interpolated values of the velocity gradient,
4. add the sum of viscous fluxes and the effects of source terms to the previously calculated sum of fluxes.

All these loops can easily run in parallel because the memory conflicts are prevented. The computation was parallelized using OpenMP [18].

Short pieces of the code presenting the calculation of the flux and setup of the mesh are in Code listings 5.4 and 5.5. Please notice the advantage taken from the concepts of GTMesh.

Code listing 5.4 Example of one of the member functions responsible for computation. This function calculates the advective flux of the gaseous phase over one inner face. All the data needed are passed to the function as `FlowData` mapped to the left and right cells to the face and the auxiliary data mapped to the face itself. The results are then stored in the data structure mapped to the face. Similarly to this function, there are 3 more `ComputeFluxGas` functions handling the corresponding boundary conditions. The same approach is used to calculate the flux of the solid phase.

```

1  template<unsigned int Dimension, unsigned int ... Reserve>
2  inline void MultiphaseFlow<Dimension, Reserve...>::
3  ComputeFluxGas_inner( const FlowData<ProblemDimension> &leftData,
4                      const FlowData<ProblemDimension> &rightData,
5                      FaceData<ProblemDimension> &edgeData)
6  {
7
8      // Interpolation of velocity
9      auto edge_u_g = ( leftData.getVelocityGas() * edgeData.LeftCellKoeff ) +
10                     ( rightData.getVelocityGas() * edgeData.RightCellKoeff );
11
12     // Inner product of velocity and normal vector
13     double product_u_n = ( edge_u_g * edgeData.n );
14
15     // Prepare the upwind values
16     const FlowData< ProblemDimension >& faceVal = product_u_n > 0 ? leftData : rightData;
17
18     // Flux of density
19     double delta_rho = - faceVal.getEps_g() * faceVal.rho_g *
20                       product_u_n * edgeData.Measure;
21     // Add the artificial dissipation
22     delta_rho += edgeData.MeasureOverDist * artificialDissipationGas *
23                ( rightData.rho_g * rightData.getEps_g() -
24                  leftData.rho_g * leftData.getEps_g() );
25
26
27     // Computing the flux of momentum
28     Vector<ProblemDimension, double> fluxP_g = (-product_u_n) * faceVal.p_g;
29
30     // Multiply by the face/edge measure
31     fluxP_g *= edgeData.Measure;
32
33     // Add the artificial dissipation
34     fluxP_g += ( edgeData.MeasureOverDist * artificialDissipationGas *
35                ( rightData.p_g - leftData.p_g ) );
36
37     // Computation of grad_p
38     double face_p_g = leftData.getPressure() * edgeData.LeftCellKoeff +
39                      rightData.getPressure() * edgeData.RightCellKoeff;
40     edgeData.grad_p = ( edgeData.Measure * face_p_g ) * edgeData.n;
41
42     // Computation of grad(u)
43     edgeData.grad_u_g = tensorProduct( edge_u_g, edgeData.n ) * edgeData.Measure;
44
45     // Store the values of fluxes at the face
46     edgeData.fluxP_g = fluxP_g;
47
48     edgeData.fluxRho_g = delta_rho;
49
50 }

```

Code listing 5.5 Implementation of the mesh setup and data exporting member functions. The `setupMeshData` member function loads the mesh from file and stores the pointer to `MeshReader` into the prepared attribute `reader`. When the mesh is correctly loaded, the auxiliary mesh quantities are calculated. Notice the usage of the `GTMesh` interface in the calculation. The presented `setupMeshData` is applied in the 3D case, the implementation of the 2D case slightly differs, since the method need not be as general. The `exportData` member function exports the mesh and the numerical solution. Notice please how easy it is to realize the export when using the concepts of `GTMesh`. Let us note that this code even exports the data of the tessellated mesh automatically.

```

1  template<unsigned int Dimension, unsigned int ... Reserve>
2  template<unsigned int _Dimension>
3  typename std::enable_if<_Dimension == 3>::type
4  MultiphaseFlow< Dimension, Reserve... >::setupMeshData(const std::string& fileName){
5      // Load the mesh and setup the properties
6      reader = mesh.load(fileName);
7      mesh.template initializeCenters<METHOD_TESSELLATED>();
8      mesh.setupBoundaryCells();
9      mesh.setupBoundaryCellsCenters();
10
11     // Calculate the measures of the mesh elements
12     auto measures = mesh.template computeElementMeasures<METHOD_TESSELLATED>();
13
14     // Calculate the distances between neighboring cells
15     auto dists = ComputeCellsDistance(mesh);
16
17     // Calculation of the mesh properties
18     auto faceNormals = mesh.template computeFaceNormals<METHOD_TESSELLATED>();
19
20     // Resize the meshData container according to the mesh
21     meshData.allocateData(mesh);
22
23     // Store the inverted values of cells
24     for (const auto& cell : mesh.getCells()) {
25         meshData.at(cell).invVolume = 1.0 / measures.at(cell);
26     }
27
28     for(const auto& face : mesh.getFaces()) {
29         meshData.at(face).Measure = measures.at(face);
30
31         // ** Setup of the rest of the quantities calculated in advance **
32     }
33 }

```

```

1  template <unsigned int Dimension, unsigned int ... Reserve>
2  void MultiphaseFlow< Dimension, Reserve... >::
3      exportData( double time,
4                  MeshDataContainer<ResultType, ProblemDimension>& compData,
5                  double timeModifier ) {
6
7      char timeStr[20];
8      sprintf(timeStr, "%04ld", lround(time*timeModifier));
9      // Export the mesh
10     std::ofstream ofile(std::string("MultiphaseFlow") + "_" + timeStr + ".vtk");
11     writer.writeHeader(ofile, std::string("MPF_") + std::to_string(time));
12     writer.writeToStream(ofile, mesh, reader->getCellTypes());
13     // Write the result of the simulation
14     VTKMeshDataWriter<ProblemDimension> dataWriter;
15     dataWriter.writeToStream(ofile, compData, writer);
16
17     ofile.close();
18
19 }

```

Chapter 6

Parallel Implementation on GPU

In the recent years, Graphical Processing Units (GPUs) evolved into universal computational accelerators extremely efficient for massively parallel computations. Several successful product series exist that are specifically tailored toward general purpose computations on GPU (GPGPU) and, more recently, deep learning. The GPU is connected to the CPU through the PCI Express interface or a high bandwidth bus as NVLink. It has its own memory (*global memory*). The global memory bandwidth is several times faster than the RAM connected to the CPU. Unfortunately, in comparison to the speed of the attached memories, the communication between the CPU and GPU is slow. Therefore, the performance may suffer from frequent communication between CPU and GPU.

In the future work, we plan to utilize the CUDA (Compute Unified Device Architecture) framework provided by nVIDIA for programming the GPU. For more information about CUDA, see [34, 15].

In contrast to multi-core CPUs, the GPU has a *many-core* architecture. Current NVIDIA GPUs consist of up to more than a hundred *streaming multiprocessors* (SM). Each SM has a hierarchical internal structure, containing many specialized processing units (32/64 bit CUDA cores, Tensor cores etc.) designed for parallel processing of certain mathematical operations in a SIMD (Single Instruction Multiple Data) fashion [44]. From the software point of view, the hierarchical architecture of the GPU reflects in the grouping of the individual computational "CUDA" threads into SIMD units (so called *warps*).

The units within one SM share a small (tens of kilobytes) but very fast memory in comparison to the global memory. The SMs can not communicate with each other, but they can access the global memory. In order to maximize the computational efficiency, it is recommended to coalesce the accesses into the global memory. This requires optimizing the data structures for use on the GPU. In connection with the data organization, we want to develop a container automatically rearranging the data between an array of structures (AoS) and a structure of arrays (SoA) (see Section 6.3).

Although the GPU has thousands of CUDA cores, it requires to run much more (hundred thousands) threads at once in order to maximize performance. This is completely opposite to the practices on CPU where additional threads degrade performance.

In order to adapt GTMesh to the GPU, it was planned to cooperate with the TNL project [22] and utilize their containers able to allocate memory on either CPU or GPU depending on the template setup. There are three steps to adapt the whole computation to the GPU:

1. Adapt the unstructured mesh itself,
2. move the data mapped to the mesh (`MeshDataContainer`) to the GPU memory,
3. perform the RKM (see Section 4.3) algorithm on the GPU, as described in detail in [46].

A possible approach of computing on unstructured grid on the GPU can be found in [51, 54]

Code listing 6.1 Changes in MeshDataContainer in order to utilize the TNL::Containers::Vector container instead of std::vector.

```

1  template<typename DataType, typename DeviceType, unsigned int MappedDimension = 0>
2  struct DataContainer : public TNL::Containers::Vector<DataType, DeviceType> {
3      using type = DataType;
4
5      static constexpr unsigned int getMappedDimension() {
6          return MappedDimension;
7      }
8  };
9
10
11 template <typename DataType, typename DeviceType, unsigned int ...Dimensions>
12 struct MeshDataContainer{
13 private:
14     template<typename _DataType, unsigned int Pos>
15     struct _DataContainer : _DataContainer<_DataType, Pos - 1> {
16         DataContainer<_DataType, DeviceType, dimensionAt<Pos>()> _data;
17     };
18
19     template<typename _DataType>
20     struct _DataContainer<_DataType, 0>{
21         DataContainer<_DataType, DeviceType, dimensionAt<0U>()> _data;
22     };
23
24     _DataContainer<DataType, sizeof... (Dimensions) - 1> data;
25
26 // ... ** member functions **
27 // the implementation must reflect the TNL::Containers::Vector interface
28 };

```

6.1 Adaptation of UnstructuredMesh to GPU

Thanks to the fact that MeshElements consists of several arrays containing simple MeshElement structures, it is relatively easy to move this complete structure to the GPU. However, as discussed at the beginning of this chapter, coalesced memory access is required to process the data structure efficiently. Additionally, most of the implemented algorithms such as mesh import and export can be performed on the CPU only.

Therefore, we prefer to store the whole mesh on the CPU and export only the data necessary for the computation to the GPU. The necessary data are the computational data such as the unknown variables, and the auxiliary mesh data, e.g., measures of the elements or distances between cell centers. Finally, the topology of the mesh can be ported to the GPU as the connections (see Section 2.4.2). For example, in the two-phase problem, the requested connections were from cells to faces and opposite. This approach also benefits from the fact that the data on the GPU are limited to the needs of the problem which saves the global memory which has a limited capacity with respect to the RAM size of the host, e.g., would be not necessary to pass the mesh vertices to the GPU in the case of the two-phase flow problem.

Alternatively, it is possible to utilize sparse matrix formats to store the connections in the form of adjacency matrices [54]. The advantage of this approach consists in the optimality of the sparse matrix formats on the GPU.

6.2 Adaptation of MeshDataContainer to GPU

As was said in the previous Section 6.1, it is necessary to move the computational data (i.e., MeshDataContainer) to the GPU. In this section the steps required to adapt MeshDataContainer to the GPU using TNL::Containers::Vector are presented.

The first step is to substitute the std::vector inheritance of the DataContainer (see Code listing 2.24) for TNL::Containers::Vector and add the template argument Device specifying

the device (CPU or GPU) to be used for data storage (see Code listing 6.1). The device argument must also be added to the template argument list of `MeshDataContainer`.

As it is not possible to pass the variables through references into CUDA kernels, TNL utilizes the system of non-owning containers called *views*, e.g., `TNL::Containers::VectorView`. A view has minimum information about the array, such as pointer to the data and the size of the array, and in its copy constructor it creates a shallow copy only. Therefore, a view of a container can be passed to a kernel by value. The views in TNL can be assigned to the container by the `bind` member function.

To fully adapt the `MeshDataContainer` to the GPU under the terms of TNL, the `MeshDataContainerView` is to be implemented. The fundamentals of the implementation of the `MeshDataContainerView` are presented in Code listing 6.2.

6.3 Automatic Conversion Between AoS and SoA

In this work, arrays of structures (AoS) in `MeshDataContainer` are utilized in the form of `std::vector` or `TNL::Containers::Vector` described in this chapter. Especially, when discussing the efficiency of a computation performed on the GPU, the data organization appears to be a significant factor as discussed in [50]. For the basic example of the difference between the SoA and AoS layouts, see Code listing 6.3

Utilizing the concept of traits (especially `DefaultArithmeticTraits`), it is possible to create class template automatically generating the structure of arrays from a traitled class (see Table 3.1). The class understands the content of the given traitled class and creates a system of arrays containing the corresponding types. Moreover, it is possible to recursively continue with the parsing of the traitled class, i.e., the given traitled class can have an attribute of another traitled class. Finally, it is possible to unify the interface of SoA and AoS (`TNL::Containers::Vector`), so that the user can treat both data layouts in the same way. However, this concept is very complex and beyond the scope of this thesis.

Code listing 6.2 Implementation of the concept of views from TNL [22] in MeshDataContainer.

```

1  template<typename DataType, typename DeviceType, unsigned int MappedDimension = 0>
2  struct DataContainerView : public TNL::Containers::VectorView<DataType, DeviceType> {
3      using type = DataType;
4
5      static constexpr unsigned int getMappedDimension() {
6          return MappedDimension;
7      }
8  };
9
10
11 template <typename DataType, typename DeviceType, unsigned int ...Dimensions>
12 struct MeshDataContainerView{
13     using ViewType = MeshDataContainerView<DataType, DeviceType, Dimensions...>;
14 private:
15     template<typename _DataType, unsigned int Pos>
16     struct _DataContainerView : _DataContainer<_DataType, Pos - 1> {
17         DataContainerView<_DataType, DeviceType, dimensionAt<Pos>()> _data;
18     };
19
20     template<typename _DataType>
21     struct _DataContainerView<_DataType, 0>{
22         DataContainerView<_DataType, DeviceType, dimensionAt<0U>()> _data;
23     };
24
25     _DataContainerView<DataType, sizeof... (Dimensions) - 1> data;
26     ViewType getView(){
27         ViewType res;
28         res.bind(*this);
29         return res;
30     }
31 // ... ** member functions **
32 // Have similar interface to MeshDataContainer (e.g. getDataByPos, getDataByDim, etc.)
33 // the implementation must reflect the TNL::Containers::Vector interface
34 };
35
36 template <typename DataType, typename DeviceType, unsigned int ...Dimensions>
37 struct MeshDataContainer{
38     using ViewType = MeshDataContainerView<DataType, DeviceType, Dimensions...>;
39 private:
40
41     template<unsigned int Index = 0>
42     typename std::enable_if<(Index < sizeof(Dimensions...))>::type
43     bindView(ViewType& view){
44         view.template getDataByPos<Index>().bind(this->getDataByPos<Index>().getView());
45         bindView<Index + 1>(view);
46     }
47
48     template<unsigned int Index = 0>
49     typename std::enable_if<(Index == sizeof(Dimensions...))>::type
50     bindView(ViewType& view){
51         view.template getDataAtPos<Index>().bind(this->getDataAtPos<Index>().getView());
52     }
53 public:
54     ViewType getView(){
55         ViewType view;
56         bindView(view);
57         return view;
58     }
59 // ... ** member functions **
60 // the implementation must reflect the TNL::Containers::Vector interface
61 };

```

Code listing 6.3 Difference between the AoS and SoA data layouts. Whereas the AoS layout lies the attributes `r`, `g`, `b` in a row, the SoA creates the array containing only the `r` attribute followed by the arrays containing `g` and `b` attributes. Let us point out the difference between the element access in SoA and AoS layouts. In the AoS layout the the `i`th element is accessed first and then the requested member. In the SoA layout, the array containing the instances is accessed first, the `i`th element is accessed as second.

```
1  constexpr int N;
2  struct ColorAoS{
3      char r;
4      char g;
5      char b;
6  } ArrayOfStructures[N];
7
8  struct ColorSoA{
9      char r[N];
10     char g[N];
11     char b[N];
12 } StructureOfArray;
13
14 // Example of accessing the r member at the position i
15 ArrayOfStructures[i].r;
16 StructureOfArray.r[i];
```

Chapter 7

Simulations

In this chapter, the results of simulations are presented. The first discussed case is a comparison of gas flow simulations performed on different meshes. The next simulation is two-phase flow in a 2D domain representing a combustion chamber of a fluidized bed (FB) boiler [26, 27]. The final simulation represents two-phase flow of gas and sand in the combustion chamber in 3D.

In comparison to the previous work [40], the simulation was extended from 2D to 3D. At the beginning, the simulation of heat conduction on an unstructured 3D mesh was performed to check that the GTMesh library works properly. Then, during the implementation of two-phase flow in 2D and 3D, the results were compared to the results obtained by the previous 2D implementation. This way, the correct function of the `MultiphaseFlow` structure was checked. Thanks to the template implementation of `MultiphaseFlow`, the numerical scheme is independent of dimension. Hence, it is easy to switch the computation from 2D to 3D.

7.1 Comparison of Gas Flow on Several Meshes

This section presents the results of single phase flow performed on 5 different meshes tessellating the same geometry shown in Figure 7.2.

Three of the meshes are structured ($\mathcal{T}_{s,1}$, $\mathcal{T}_{s,2}$, $\mathcal{T}_{s,3}$) and two are unstructured ($\mathcal{T}_{u,1}$, $\mathcal{T}_{u,2}$). The unstructured meshes were generated by reflecting the cubes presented in Figure 7.1. The structured meshes employ a uniform subdivision of the domain into cubes. See the properties of the meshes in Table 7.2. The parameters settings are summarized in Table 7.1. The results are presented in Figures 7.3, 7.4 and 7.5.

The results on the structured meshes do not differ much for different granularities of the mesh because the structured meshes with faces either perpendicular or parallel to the flow suppress the numerical dissipation which is inherent to first order finite volume schemes [30]. On unstructured meshes, the numerical diffusion is much more significant. The numerical dissipation also causes the time step of the Runge-Kutta-Merson method to be shortened in accordance with the stability condition [33, 35, 49]. As a result, the computational times for the unstructured meshes are longer, see Table 7.2.

At the time $t = 0.5$ s, the differences between simulations on $\mathcal{T}_{s,3}$ and $\mathcal{T}_{u,2}$ are moderate, but they as the flow stabilizes grow bigger, see the result at $t = 3$ s. Moreover, in the simulations on $\mathcal{T}_{u,1}$ artifacts can be seen. These high values appear at the boundary and they may be caused by the geometry of the cells. Such artifacts do not appear in simulation on the finer unstructured mesh $\mathcal{T}_{u,2}$.

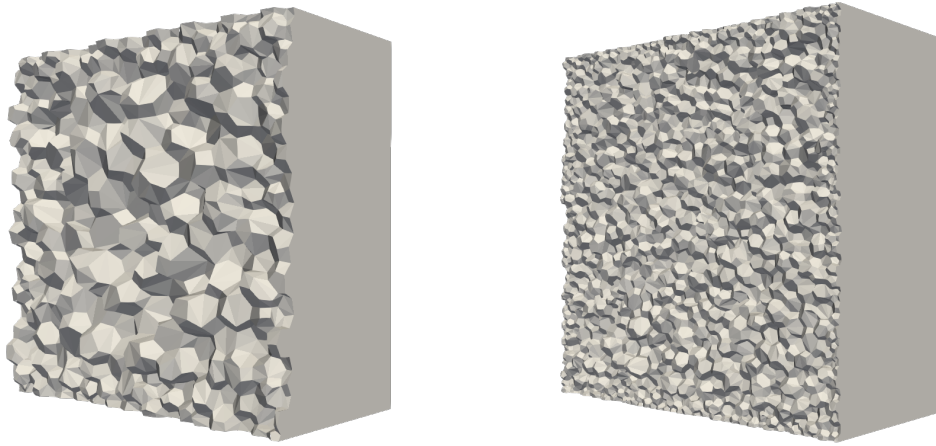


Figure 7.1: The slices of unstructured meshes tessellating a cube with edge length 2.5 meters. These meshes were utilized to create unstructured stacks in Section 7.1. The unstructured meshes are courtesy of J. Hahn / AVL Fire [2].

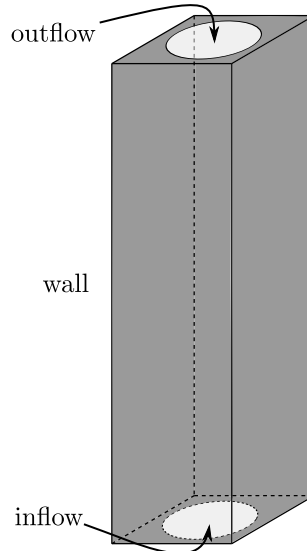


Figure 7.2: Spatial domain representing a stack for the gas flow case. The cuboid is 10 meters high, 2.5 meters wide and 2.5 meters deep.

quantity/parameter	value
T	300 K
μ_g	$10^{-5} \text{ N} \cdot \text{s} \cdot \text{m}^{-1}$
R_{spec}	$287 \text{ J} \cdot \text{K}^{-1} \cdot \text{kg}^{-1}$
$p_{g,\text{out}}$	10^5 Pa
$\mathbf{u}_{g,\text{in}}$	$\begin{pmatrix} 0 \\ 0 \\ (x_1^2 + x_2^2 - 0.09) \frac{1}{0.09} \end{pmatrix} \text{ m} \cdot \text{s}^{-1}$
$\mathbf{u}_{g,\text{ini}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$
$p_{g,\text{ini}}$	10^5 Pa
\mathbf{g}	$\begin{pmatrix} 0 \\ 0 \\ -9.81 \end{pmatrix} \text{ m} \cdot \text{s}^{-2}$
λ_g	0.0

Table 7.1: Parameter settings for the flow of gas in the stack. The parameters $\varepsilon_{s,\text{ini}}$ and $\varepsilon_{s,\text{in}}$ are set to zero in order to simulate single phase flow. The results of the computation are shown in Figures 7.3, 7.4, 7.5.

mesh	#cells	#faces	spatial resolution	time [s]	RKM time [ms]	RKM iterations
$\mathcal{T}_{s,1}$	10976	34692	$14 \times 14 \times 56$ cells	50	9.9	5028
$\mathcal{T}_{s,2}$	108000	332100	$30 \times 30 \times 120$ cells	315	43	7238
$\mathcal{T}_{s,3}$	237276	725517	$39 \times 39 \times 156$ cells	763	86	8867
$\mathcal{T}_{u,1}$	16316	105834	-	2520	26	98917
$\mathcal{T}_{u,2}$	128016	859966	-	23800	106	224252

Table 7.2: Parameters of the utilized meshes and the time consumption of the simulations. Let us note that the number of faces also influences the performance of the computation because of the iterations over cells boundaries. The fifth column presents the total time the simulations took in seconds. The time demand depends on many factors. One of them is the average time consumed by of a single update calculated by RKM method (the sixth column). This is especially influenced by the number of elements of the mesh. Another factor is the total number of RKM iterations needed to calculate the simulation (the seventh column). Each simulation was performed on one compute node of the the HELIOS cluster at the Department of Mathematics, FNSPE CTU Prague. Each node is equipped with two 16-core AMD EPYC 7281@2.1GHz CPU (SMT mode disabled) and 128 GB RAM. OpenMP parallelization with 32 threads was employed.

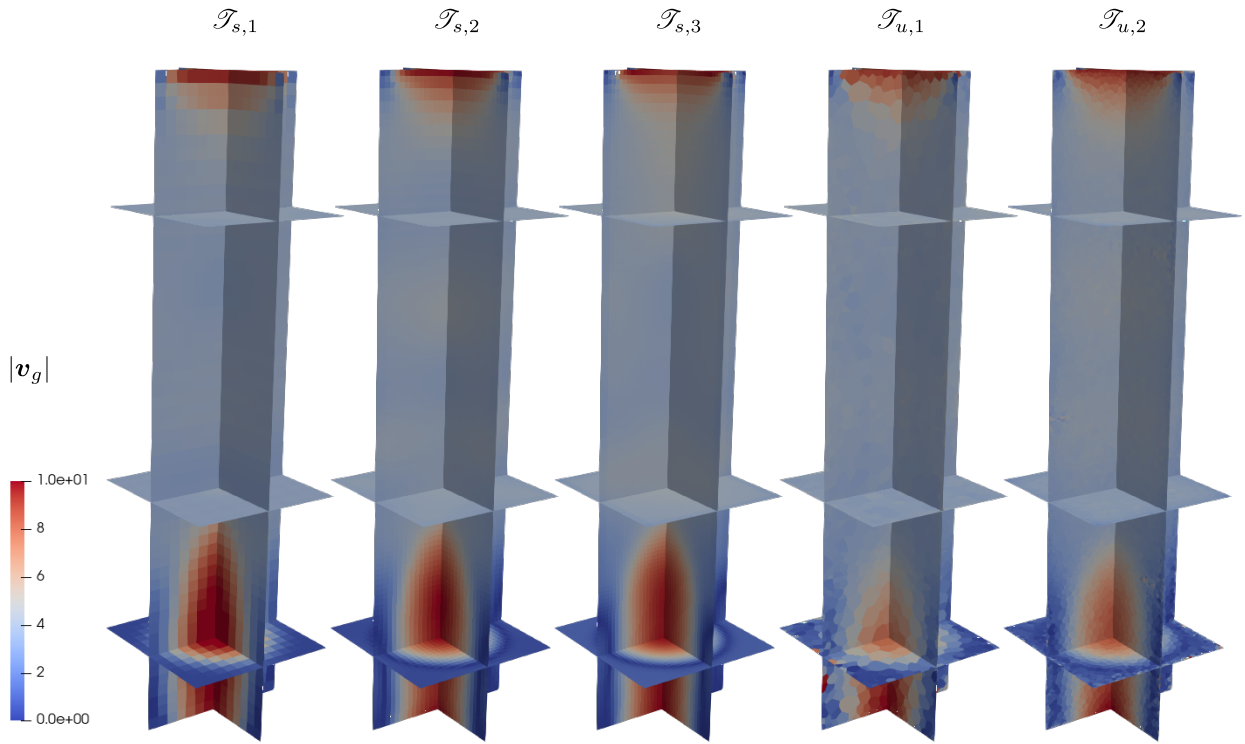


Figure 7.3: The magnitudes of velocity in $\text{m} \cdot \text{s}^{-1}$ at time $t = 0.5$ s obtained by computations of gas flow on the selected meshes. The domain is presented in Figure 7.2. The parameters settings of the simulation are summarized in Tables 7.1, 7.2.

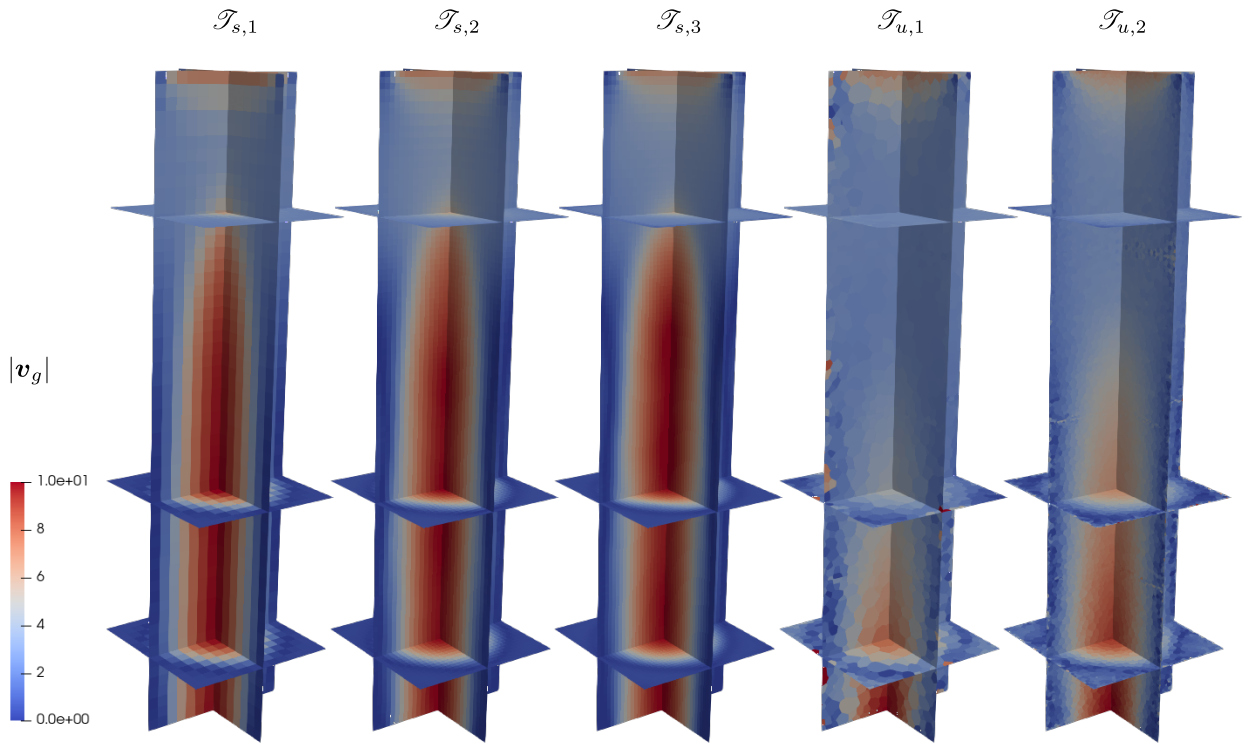


Figure 7.4: The magnitudes of velocity in $\text{m} \cdot \text{s}^{-1}$ at time $t = 1.5$ s obtained by computations of gas flow on the selected meshes. The domain is presented in Figure 7.2. The parameters settings of the simulation are summarized in Tables 7.1, 7.2.

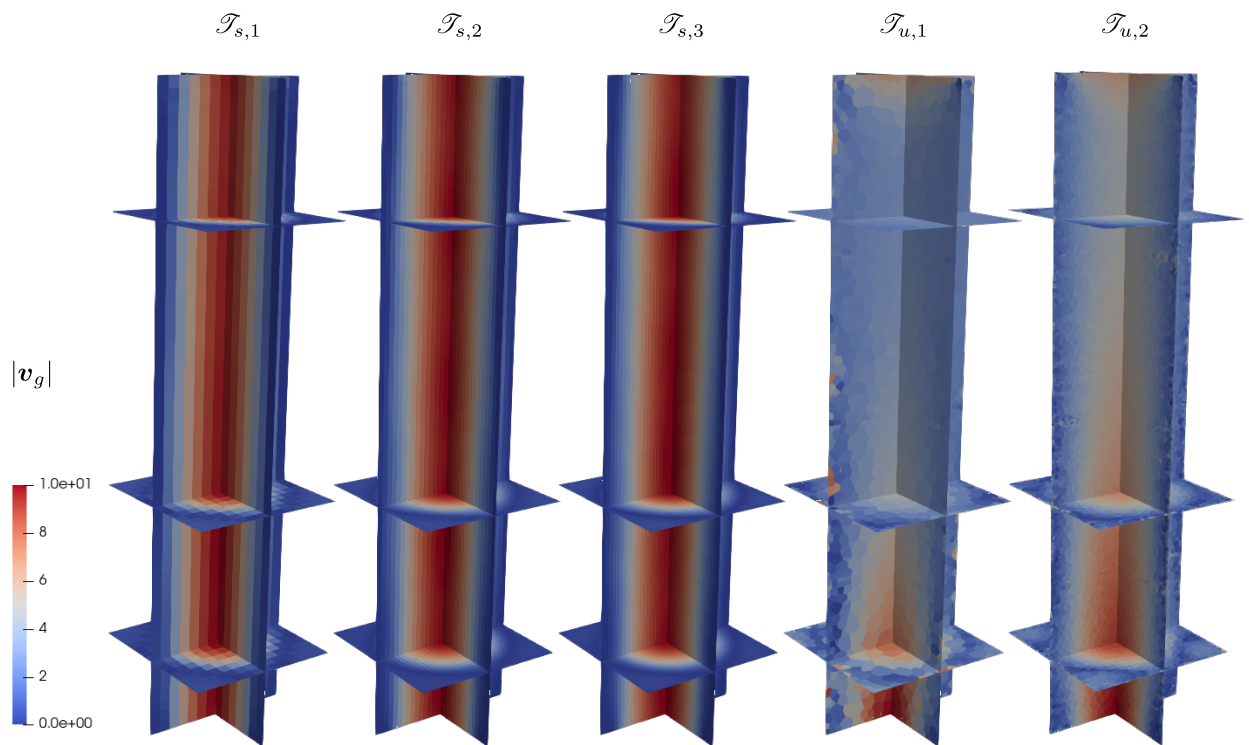


Figure 7.5: The magnitudes of velocity in $\text{m} \cdot \text{s}^{-1}$ at time $t = 3$ s obtained by computations of gas flow on the selected meshes. The domain is presented in Figure 7.2. The parameters settings of the simulation are summarized in Tables 7.1, 7.2.

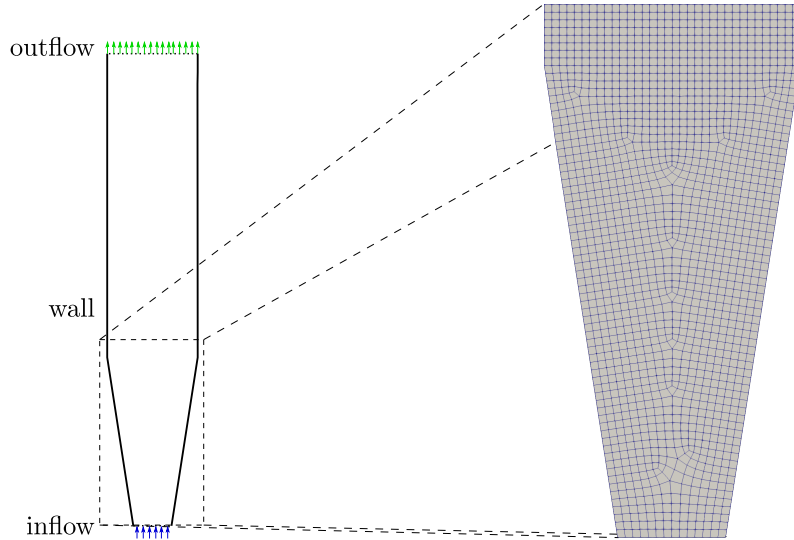


Figure 7.6: Computational domain representing a 2D model of the combustion chamber of a FB boiler with inflow at the bottom and outflow at the top. The boiler domain is 34 meters high and 7.6 meters wide. On the right, the portion of the used mesh is presented.

7.2 Two-Phase Flow in 2D

This simulation represents a two-phase flow, where one phase is gaseous and the other is granular solid (sand) in a 2D domain. The domain setup is shown in Figure 7.6. The time interval is $\mathcal{I} = (0, 30)$ in seconds.

The initial state is zero velocity for both phases and a 12 meters high dispersed pile of sand ($\varepsilon_s = 0.2$) placed two meters above the inflow. The initial pressure is set to 10^5 Pa. At the part of boundary denoted as inflow, the wall boundary condition is prescribed for the solid phase in order to prevent the escape of solid phase through the inflow. The granular solid phase is considered to have density of $1700 \text{ Kg}\cdot\text{m}^{-3}$ and spherical grains ($\phi_s = 1$) with diameter 0.78 mm. The complete parameter setup is presented in Table 7.3.

At the beginning, the whole pile of solid phase is hovered by the growing pressure below. Then, most of the mass is split and thrown at the walls (Figure 7.7). The further evolution continues by bubbling and creating whirls (Figures 7.8 and 7.9) while the mass of the solid phase resides in the lower half of the computational domain. The overall maximum ε_s in any single cell did not exceed the value of 0.5.

The simulation was also performed with lower artificial diffusion. The lower diffusion causes shortening of the time step in the RKM solver. The utilized model does not reflect the effect of turbulence, therefore the results with higher artificial diffusion may be more realistic.

quantity/parameter	value	quantity/parameter	value
T	300 K	ρ_s	$1700 \text{ Kg} \cdot \text{m}^{-3}$
μ_g	$10^{-5} \text{ N} \cdot \text{s} \cdot \text{m}^{-1}$	μ_s	$0.5 \text{ N} \cdot \text{s} \cdot \text{m}^{-1}$
R_{spec}	$287 \text{ J} \cdot \text{K}^{-1} \cdot \text{kg}^{-1}$	ϕ_s	1
$p_{g,\text{out}}$	10^5 Pa	d_s	$7.8 \cdot 10^{-4} \text{ m}$
$\mathbf{u}_{g,\text{in}}$	$\begin{pmatrix} 0 \\ 4 \end{pmatrix} \text{ m} \cdot \text{s}^{-1}$	$\mathbf{u}_{s,\text{in}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$
$\mathbf{u}_{g,\text{ini}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$	$\mathbf{u}_{s,\text{ini}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$
$p_{g,\text{ini}}$	10^5 Pa	$\varepsilon_{s,\text{ini}}$	$\begin{cases} 0.2 & x_2 \in (2, 14) \\ 0 & \text{elsewhere} \end{cases}$
\mathbf{g}	$\begin{pmatrix} 0 \\ -9.81 \end{pmatrix} \text{ m} \cdot \text{s}^{-2}$	$\varepsilon_{s,\text{in}}$	0
λ_g	0.1	λ_s	0.1

Table 7.3: Parameter settings of the two-phase flow in the combustion chamber of a FB boiler. The coordinates specifying the region of nonzero $\varepsilon_{s,\text{ini}}$ are in meters. The results of the simulation at times 5, 10 and 30 seconds are presented in Figures 7.7 - 7.9.

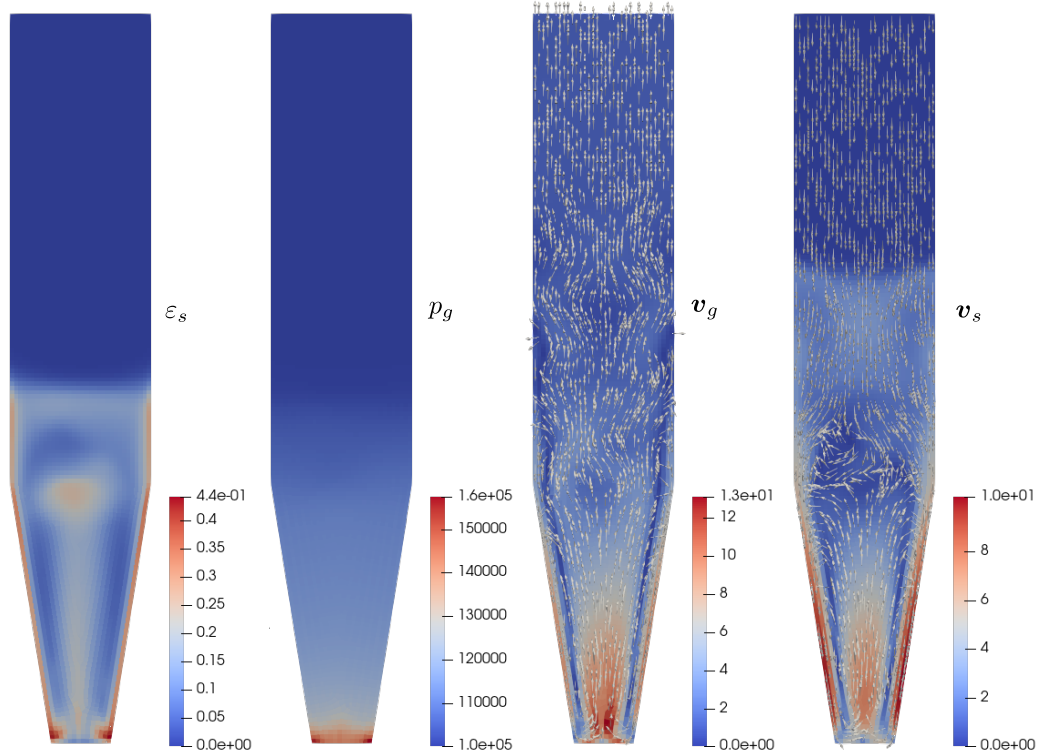


Figure 7.7: The numerical solution of two-phase flow in a 2D domain (Figure 7.6), $t = 5$ s. The parameter setup is in Table 7.3. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

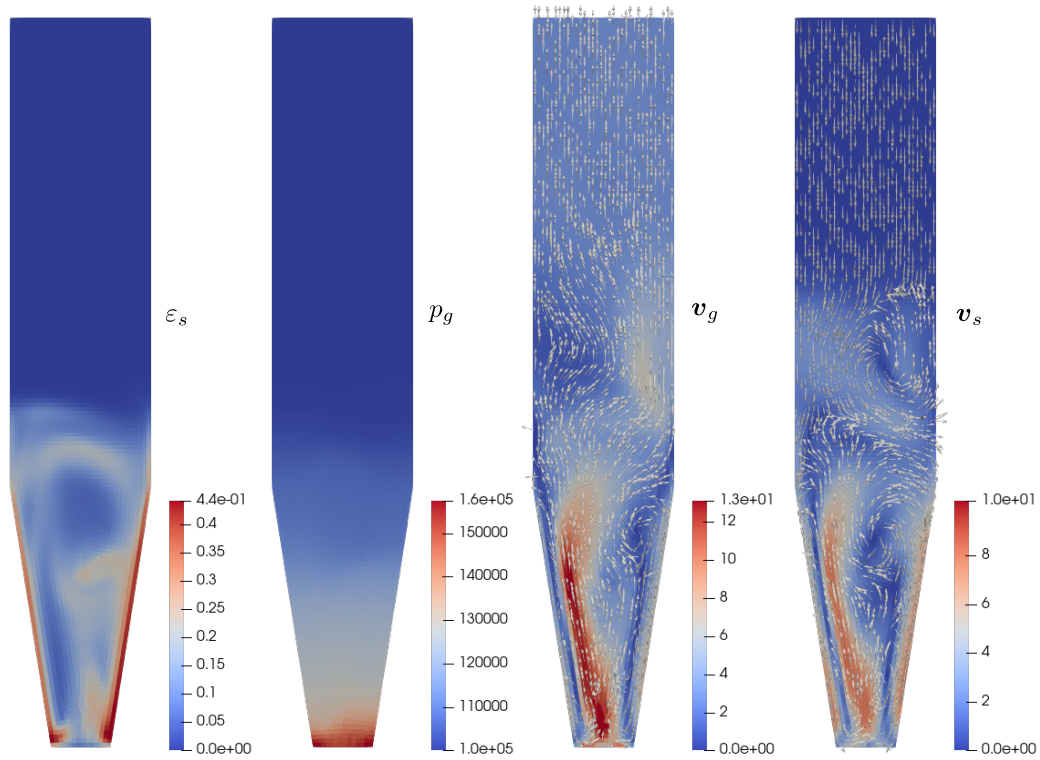


Figure 7.8: The numerical solution of two-phase flow in a 2D domain (Figure 7.6), $t = 10$ s. The parameter setup is in Table 7.3. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

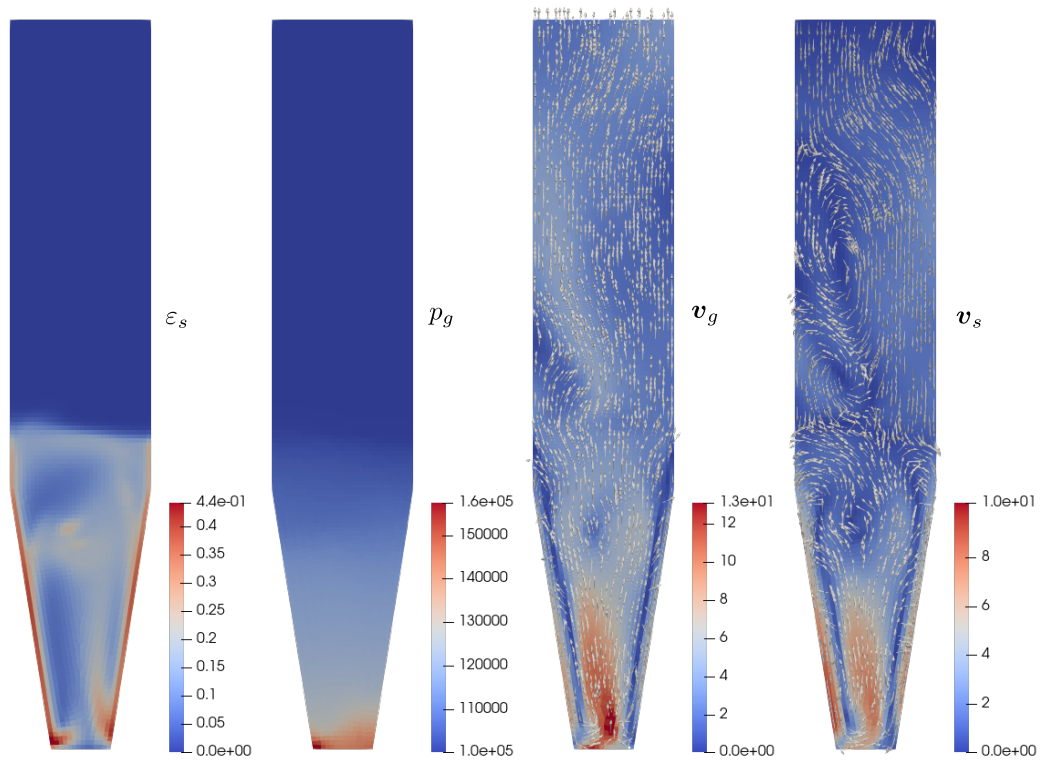


Figure 7.9: The numerical solution of two-phase flow in a 2D domain (Figure 7.6), $t = 30$ s. The parameter setup is in Table 7.3. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

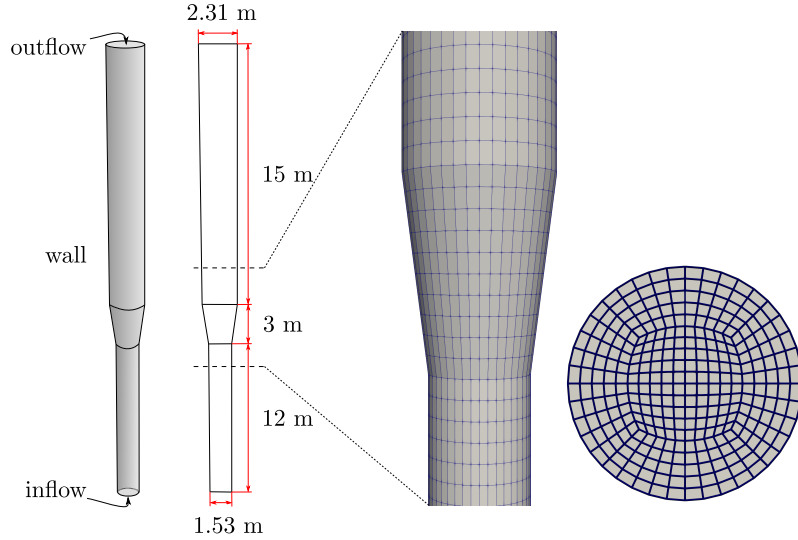


Figure 7.10: Computational domain representing a 3D model of the combustion chamber of a FB boiler with inflow at the bottom and outflow at the top. On the left, the geometry and dimensions of the domain are shown. On the right, the utilized mesh is presented. The domain geometry and mesh are based on [29, 47].

7.3 Two-Phase Flow in 3D

This section presents a two-phase flow of gas and solid in a 3D computational domain. The domain setup is shown in Figure 7.10. The time interval is $\mathcal{I} = (0, 30)$ in seconds.

The initial state is zero velocity for both phases and a 6 meters high dispersed pile of sand ($\varepsilon_s = 0.4$) placed two meters above the inflow. The initial pressure is set to 10^5 Pa. Similarly to the previous simulation (Section 7.2), at the part of boundary denoted as inflow, the wall boundary condition is prescribed for the solid phase in order to prevent the escape of solid phase through the inflow. The granular solid phase is considered to have the density of $1700 \text{ Kg} \cdot \text{m}^{-3}$ and spherical grains ($\phi_s = 1$) with diameter 0.78 mm. The complete parameter setup is presented in Table 7.4. Let us note that the axis pointing up is \mathbf{y} .

At the beginning, the whole pile of solid phase is hovered by the growing pressure below. The solid phase stops in the widening part of the boiler and settles at the walls (Figure 7.11). As the solid phase falls, it clogs the pipe up which results in creating a bubble in the middle (Figure 7.12). Then, most of the solid phase is piled at one side of the boiler and falls down along the wall (Figure 7.13).

quantity/parameter	value	quantity/parameter	value
T	300 K	ρ_s	$1700 \text{ Kg} \cdot \text{m}^{-3}$
μ_g	$10^{-5} \text{ N} \cdot \text{s} \cdot \text{m}^{-1}$	μ_s	$0.5 \text{ N} \cdot \text{s} \cdot \text{m}^{-1}$
R_{spec}	$287 \text{ J} \cdot \text{K}^{-1} \cdot \text{kg}^{-1}$	ϕ_s	1
$p_{g,\text{out}}$	10^5 Pa	d_s	$7.8 \cdot 10^{-4} \text{ m}$
$\mathbf{u}_{g,\text{in}}$	$\begin{pmatrix} 0 \\ 3 \\ 0 \end{pmatrix} \text{ m} \cdot \text{s}^{-1}$	$\mathbf{u}_{s,\text{in}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$
$\mathbf{u}_{g,\text{ini}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$	$\mathbf{u}_{s,\text{ini}}$	$\vec{0} \text{ m} \cdot \text{s}^{-1}$
$p_{g,\text{ini}}$	10^5 Pa	$\varepsilon_{s,\text{ini}}$	$\begin{cases} 0.4 & x_2 \in (2, 8) \\ 0 & \text{elsewhere} \end{cases}$
\mathbf{g}	$\begin{pmatrix} 0 \\ -9.81 \\ 0 \end{pmatrix} \text{ m} \cdot \text{s}^{-2}$	$\varepsilon_{s,\text{in}}$	0
λ_g	0.05	λ_s	0.05

Table 7.4: Parameter settings of the two-phase flow in the 3D combustion chamber of a FB boiler (Figure 7.10). The coordinates specifying the region of nonzero $\varepsilon_{s,\text{ini}}$ are in meters. The results of the simulation at times 5, 10 and 30 seconds are presented in Figures 7.11 , 7.12, 7.13.

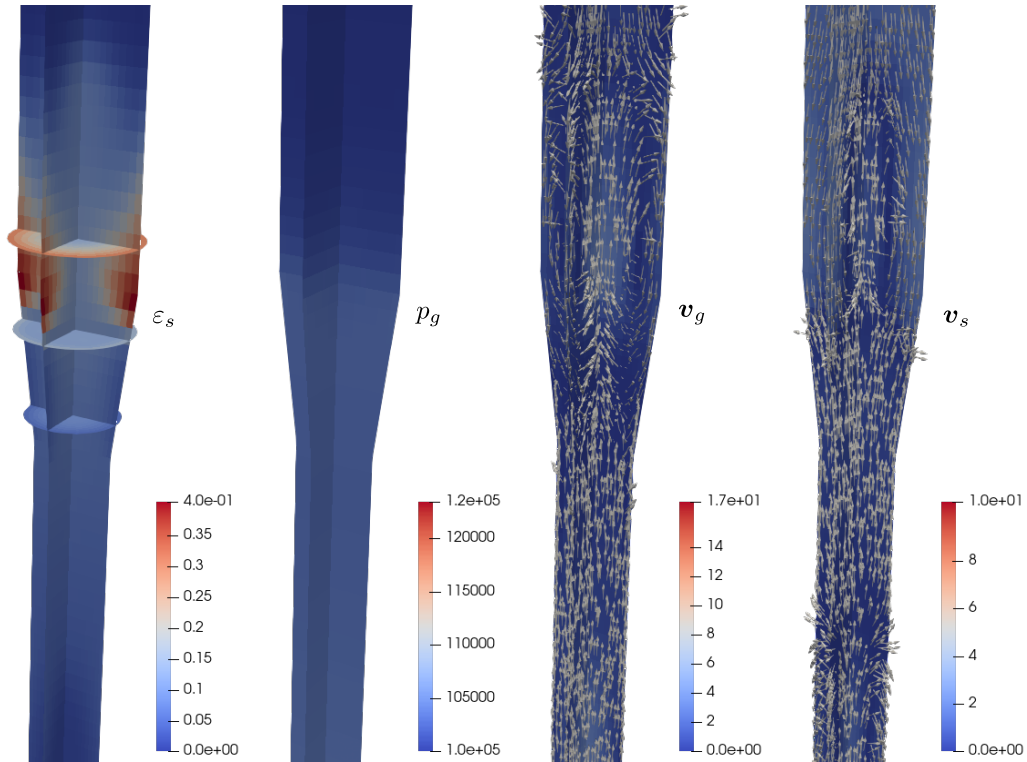


Figure 7.11: The numerical solution of two-phase flow in a 3D domain (Figure 7.10), $t = 5 \text{ s}$. The parameter setup is in Table 7.4. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

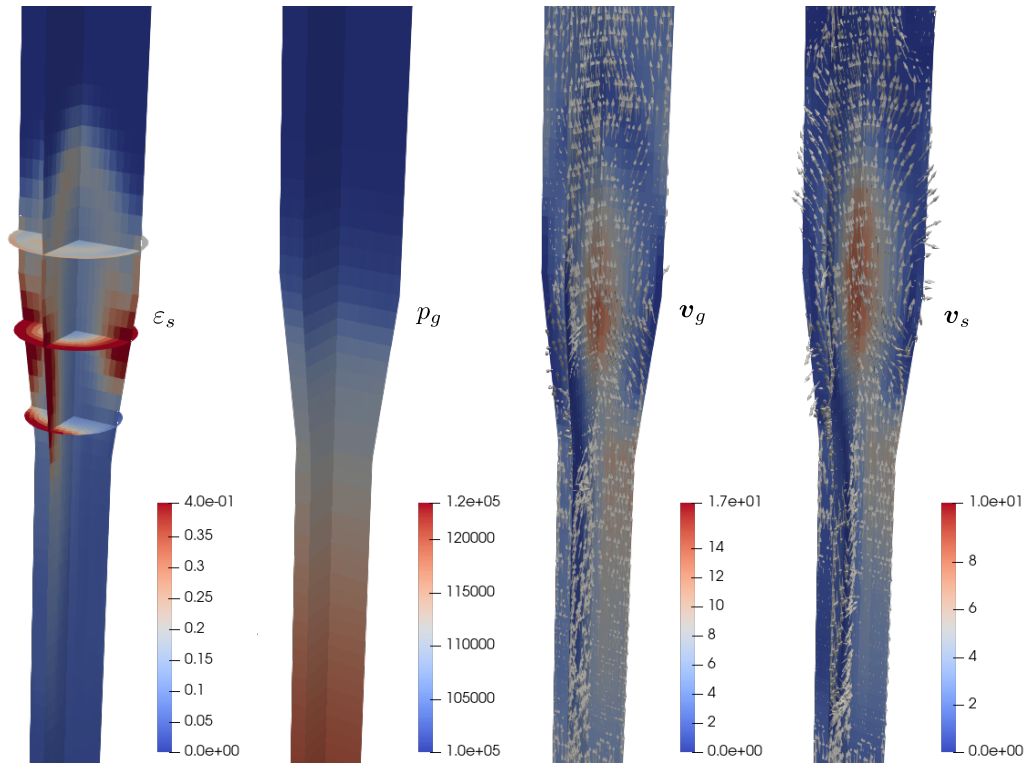


Figure 7.12: The numerical solution of two-phase flow in a 3D domain (Figure 7.10), $t = 10$ s. The parameter setup is in Table 7.4. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

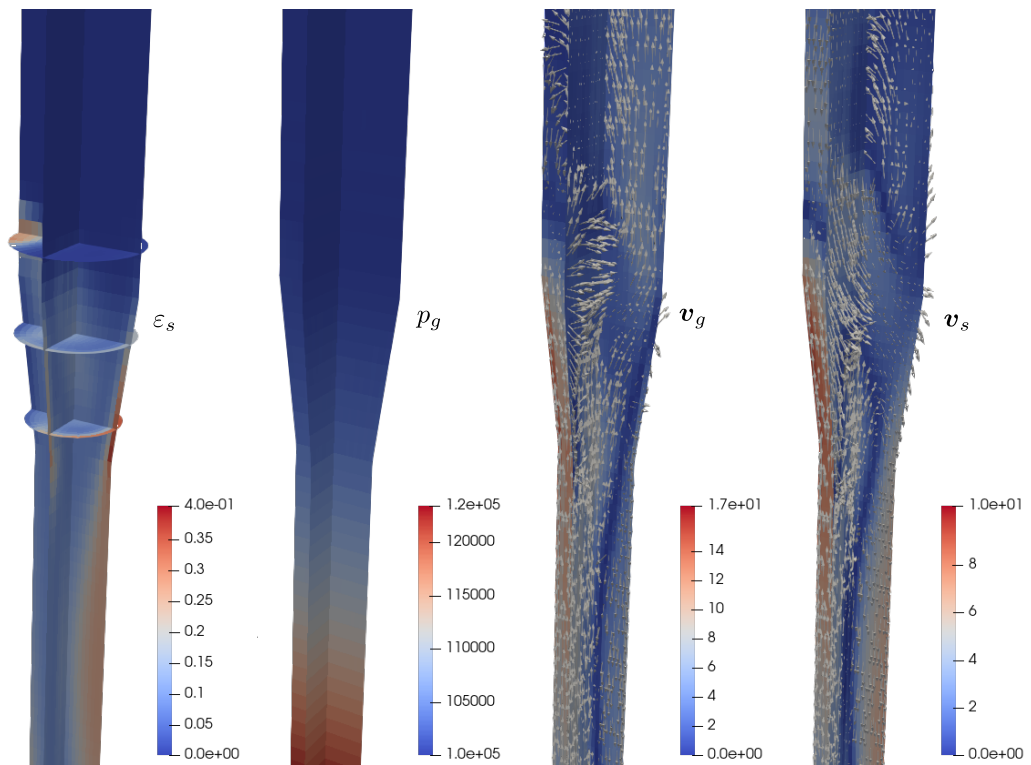


Figure 7.13: The numerical solution of two-phase flow in a 3D domain (Figure 7.10), $t = 30$ s. The parameter setup is in Table 7.4. The arrows in the two figures on the right represent the direction of the corresponding velocities while the color scales represent their magnitudes.

Conclusion

At the beginning of this thesis, a formalism of a graph description of unstructured meshes was introduced. Similar descriptions are presented by other authors in [28, 55], nevertheless their formalism is closer to description of connections between elements in the mesh than to the graph theory. Therefore, the approach presented in this work is an abstraction of the results from [28, 55] utilizing the tools of graph theory. Using the developed formalism, the data structures storing unstructured meshes were discussed.

Then, a C++ numerical library (*GTMesh*) working with unstructured meshes with a general topology and dimension was developed. This thesis provides a detailed description of *GTMesh* algorithms and project architecture. Emphasis was put on the explanation of the chosen implementation and why we decided to realize the functionality in such a way.

Despite its generality, *GTMesh* is an easily maintainable and extendable project thanks to the architecture, especially to the relation of the classes `MeshElement` and `UnstructuredMesh`. Besides the structure storing an unstructured mesh, *GTMesh* provides an easy and intuitive way of managing the data mapped to the mesh through a class named `MeshDataContainer`. `MeshDataContainer` is even capable of storing multiple data types mapping to various dimensions of the mesh. We have developed the basic mesh algorithms such as face normal or element measure calculation together with the algorithms determining the neighborhood of elements or the proper coloring of the mesh.

To achieve the generality, *GTMesh* utilizes modern C++ paradigms (template metaprogramming, SFINAE, ADL, etc.) [3, 5, 7]. *GTMesh* proves itself as a user friendly and efficient framework for creating custom numerical algorithms utilizing, for example, the finite element method or the finite volume method (FVM). Using almost every tool in *GTMesh* requires only one line of code. Therefore, the programming with *GTMesh* looks more like scripting. For example, measures of all elements of the mesh can be obtained by calling the `calculateElementMeasures` member function.

GTMesh also provides a system annotating the attributes of C++ structures called class traits, which is a generalization of a similar C++ project [21]. The class traits are utilized in a general purpose logging tool which is able to log almost any C++ expression. The class traits also enabled us to develop a system of automatic export and import of the computational data, e.g., in VTK file format [23].

The logging tool was used to comfortably check the functionality of the library during the development in the form of manual tests. Then, we utilized the approach of continuous integration [10] and transformed the manual tests into unit tests using the *GTest* library [11].

Thanks to optimizations, the class traits provide the same efficiency of data access as the explicitly written code. Therefore, it was possible to define arithmetic functions and operators working with the traited classes and use them in numerical algorithms. This makes the resulting code more readable and maintainable. The use of the data structures also helps to improve spatial location of the data in memory. This is another factor improving the performance of the numerical algorithms.

From the beginning, *GTMesh* was developed with respect to the possibility of further evolution and its utilization in parallel computing, by means of, e.g., OpenMP, MPI or CUDA (described in Chapter 6), as a part of TNL project [22]. In comparison, in ANSYS Fluent [1]

or OpenFOAM [17] the attempts to adapt the code to GPU have been unsuccessful. So far, we were able to implement the numerical algorithms using OpenMP and to utilize the developed logging tool on GPU with CUDA framework.

To demonstrate a successful numerical solver based on GTMesh, we have presented a problem of two-phase flow in 2D and 3D. An effort was made to describe the complete application of the finite volume method on the problem on a general unstructured mesh. The discussion of the application of FVM also includes the case of 3D unstructured meshes with non-planar faces and the optimization of computation on such geometry.

Eventually, the numerical solution of equations obtained by the application of FVM was implemented using the GTMesh framework on 2D and 3D unstructured meshes. The generality of `UnstructuredMesh` enabled us to create one implementation for both dimensions where the dimension can be chosen by a single template parameter. Let us note that the numerical scheme was developed to be the same on 3D meshes with planar and non-planar faces.

The future evolution of GTMesh can proceed in several directions. The implementation of some mesh algorithms currently limited to 2D and 3D can be extended to a general dimension, which would allow, e.g., to develop numerical schemes in space-time domains. The integration with TNL and the adaptation to GPU computations were laid out in this work, whereas some parts of the implementation and the first working numerical code are still to be finished. Also, some algorithms useful for adaptive or moving meshes deserve to be implemented directly into GTMesh.

In summary, we have developed a modern, elegant, efficient and extensible library for numerical simulations on unstructured meshes and proved its functionality by implementing a numerical solver for two-phase flow.

The GIT repository with the source code of GTMesh is available at [12] under the MIT license.

Bibliography

- [1] Ansys Fluent. <https://www.ansys.com/products/fluids/ansys-fluent>. Accessed: 2020-04-06.
- [2] AVL FIRE. <https://www.avl.com/fire>. Accessed: 2020-05-18.
- [3] cppreference.com: ADL. <https://en.cppreference.com/mwiki/index.php?title=cpp/language/adl&oldid=113272>. Accessed: 2020-03-31.
- [4] cppreference.com: integer_sequence. https://en.cppreference.com/mwiki/index.php?title=cpp/utility/integer_sequence&oldid=115697. Accessed: 2020-03-31.
- [5] cppreference.com: SFINAE. <https://en.cppreference.com/mwiki/index.php?title=cpp/language/sfinae&oldid=117405>. Accessed: 2020-03-31.
- [6] cppreference.com: std::set. <https://en.cppreference.com/mwiki/index.php?title=cpp/container/map&oldid=112565>. Accessed: 2020-03-31.
- [7] cppreference.com: Template specialization. https://en.cppreference.com/mwiki/index.php?title=cpp/language/template_specialization&oldid=99273. Accessed: 2020-03-31.
- [8] cppreference.com: using declatarion. https://en.cppreference.com/mwiki/index.php?title=cpp/language/using_declaration&oldid=116066. Accessed: 2020-03-31.
- [9] Doxygen official site. <https://www.doxygen.nl/index.html>. Accessed: 2020-06-21.
- [10] GitLab CI/CD documentation. <https://docs.gitlab.com/ee/ci/README.html>. Accessed: 2020-06-21.
- [11] Google Test repository. <https://github.com/google/googletest>. Accessed: 2020-06-21.
- [12] GTMesh repository. <https://mmg-gitlab.fjfi.cvut.cz/gitlab/jakubec/GTMesh>. Accessed: 2020-06-21.
- [13] JAXB documentation. <https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/index.html>. Accessed: 2020-03-31.
- [14] Legion official site. <https://legion.stanford.edu/>. Accessed: 2020-06-22.
- [15] nvidia.com: CUDA documentation. <https://docs.nvidia.com/cuda/>. Accessed: 2020-06-05.
- [16] Open MPI official site. <https://www.open-mpi.org/>. Accessed: 2019-08-21.
- [17] OpenFOAM official site. <https://www.openfoam.com/>. Accessed: 2020-04-06.
- [18] OpenMP official site. <https://www.openmp.org/>. Accessed: 2019-08-21.

- [19] PeLeLM official site. <https://amrex-combustion.github.io/PeLeLM/>. Accessed: 2020-06-22.
- [20] Qt official site. <https://www.qt.io/>. Accessed: 2020-06-21.
- [21] ThorsSerializer repository. <https://github.com/Loki-Astari/ThorsSerializer/blob/master/doc/full.md>. Accessed: 2020-03-31.
- [22] The TNL library. <https://tnl-project.org/>. Accessed: 2020-06-14.
- [23] Visualization Toolkit VTK file format. <https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf>. Accessed: 2019-08-19.
- [24] Ľubomíra Balková. *Lineární algebra 2, skripta*. Nakladatelství ČVUT, 2014.
- [25] Prabir Basu. *Combustion and gasification in fluidized beds*. CRC press, 2006.
- [26] Prabir Basu and James Butler. Studies on the operation of loop-seal in circulating fluidized bed boilers. *Applied energy*, 86(9):1723–1731, 2009.
- [27] Prabir Basu, Cen Kefa, and Louis Jestin. *Boilers and burners: design and theory*. Springer Science & Business Media, 2012.
- [28] Mark W Beall and Mark S Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
- [29] Michal Beneš, Pavel Eichler, Jakub Klinkovský, Miroslav Kolář, Jakub Solovský, Pavel Strachota, and Alexandr Žák. Numerical simulation of fluidization for application in oxyfuel combustion. *Discrete & Continuous Dynamical Systems-S*, pages 1–15, 2020.
- [30] Jiri Blazek. *Computational fluid dynamics: principles and applications*. Butterworth-Heinemann, 2015.
- [31] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.
- [32] Susanne Brenner and Ridgway Scott. *The mathematical theory of finite element methods*, volume 15. Springer Science & Business Media, 2007.
- [33] John Charles Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.
- [34] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.
- [35] Jan Christiansen. Numerical solution of ordinary simultaneous differential equations of the 1st order using a method for automatic step change. *Numerische Mathematik*, 14(4):317–324, 1970.
- [36] Dimitri Gidaspow. *Multiphase flow and fluidization: continuum and kinetic theory descriptions*. Academic press, 1994.
- [37] Jooyoung Hahn, Karol Mikula, Peter Frolkovič, Matej Medl’a, and Branislav Basara. Iterative inflow-implicit outflow-explicit finite volume scheme for level-set equations on polyhedron meshes. *Computers & Mathematics with Applications*, 77(6):1639–1654, 2019.
- [38] Ansys Inc. ANSYS FLUENT theory guide, 2013.

- [39] Tomáš Jakubec. *Matematické modelování a numerická simulace procesů v průmyslových kotlích*. Bakalářská práce. KM FJFI ČVUT, 2018. Bakalářská práce.
- [40] Tomáš Jakubec. *Matematické modelování a numerická simulace procesů ve fluidních kotlích*. KM FJFI ČVUT, 2019. Výzkumný úkol.
- [41] Martinek Jan. *Fluidní kotel CFB na spalování dřevní biomasy o parametrech páry 150 t/h; 9,3MPa; 530 °C*. Diplomová práce. Fakulta strojního inženýrství VUT v Brně, 2015.
- [42] John D. Anderson Jr. *Computational fluid dynamics: the basics with applications*. 1995.
- [43] Dejan Kirda. *Matematické modelovanie viacfázového prúdenia vo fluidnom lôžku*. KM FJFI ČVUT, 2016. Diplomová práca.
- [44] Rhonny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. Nvidia Ampere Architecture In-Depth. <https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/>. Accessed: 2020-06-17.
- [45] Fadl Moukalled, Luca Mangani, Marwan Darwish, et al. *The finite volume method in computational fluid dynamics*. Springer, 2016.
- [46] Tomáš Oberhuber, Atsushi Suzuki, and Vítězslav Žabka. The CUDA implementation of the method of lines for the curvature dependent flows. *Kybernetika*, 47(2):251–272, 2011.
- [47] Farooq Sher, Miguel A Pans, Chenggong Sun, Colin Snape, and Hao Liu. Oxy-fuel combustion study of biomass fuels in a 20 kWth fluidized bed combustor. *Fuel*, 215:778–786, 2018.
- [48] Rohan Stanger, Terry Wall, Reinhold Spörl, Manoj Paneru, Simon Grathwohl, Max Weidmann, Günter Scheffknecht, Denny McDonald, Kari Myöhänen, Jouni Ritvanen, et al. Oxyfuel combustion for co2 capture in power plants. *International Journal of Greenhouse Gas Control*, 40:55–125, 2015.
- [49] Pavel Strachota. *Analysis and Application of Numerical Methods for Solving Nonlinear Reaction-Diffusion Equations*. KM FJFI ČVUT, 2012. Dissertation.
- [50] Robert Strzodka. Abstraction for AoS and SoA layout in C++. In *GPU computing gems Jade edition*, pages 429–441. Elsevier, 2012.
- [51] Jacob Waltz. Performance of a three-dimensional unstructured mesh compressible flow solver on NVIDIA Fermi-class graphics processing unit hardware. *International Journal for Numerical Methods in Fluids*, 72(2):259–268, 2013.
- [52] Wikipedia contributors. Template metaprogramming — Wikipedia, The Free Encyclopedia. "https://en.wikipedia.org/w/index.php?title=Template_metaprogramming&oldid=956146355", 2020. Accessed: 2020-03-27.
- [53] Wen-ching Yang. *Handbook of fluidization and fluid-particle systems*. CRC press, 2003.
- [54] Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. A GPU-Adapted Structure for Unstructured Grids. In *Computer Graphics Forum*, volume 36, pages 495–507. Wiley Online Library, 2017.
- [55] Min Zhou, Ting Xie, Seegyong Seol, Mark S Shephard, Onkar Sahni, and Kenneth E Jansen. Tools to support mesh adaptation on massively parallel computers. *Engineering with Computers*, 28(3):287–301, 2012.

Appendix A

Distribution of GTMesh

The GTMesh library is accessible at the GitLab repository [12] under the MIT license. The repository contains the GTMesh library itself located in `src/GTMesh` and unit tests in `src/UnitTests` which are automatically performed when the repository is pushed. The tests are performed on the server and this practice of automatic testing is called continuous integration. The tests are utilizing the GTest framework [11]. An example of basic testing of the `UnstructuredMesh` class functionality is in Code listing A.1.

This work is accompanied by a CD containing the clone of the repository of GTMesh, its documentation generated by Doxygen [9], codes computing the simulations demonstrated in Chapter 7, and the experimental implementation of the logging tool and the class traits in CUDA. The content of the CD is described in detail in Table A.1.

Code listing A.1 Example of a test case testing the functionality of the `UnstructuredMesh` class for a 2D mesh using the GTest library.

```
1 TEST( UnstructuredMesh2D_Functions_Test, basicTest )
2 {
3     using MeshType = UnstructuredMesh<2, size_t, double>;
4     MeshType mesh;
5
6     // Sets the mesh up as a simple square domain diagonally split into 2 triangles
7     square(mesh); // square mesh made of 2 triangles
8
9     // Test of the cell center calculation (see Section 2.4.5)
10    auto centers = computeCenters<METHOD_DEFAULT>(mesh);
11    std::vector<Vertex<2, double>> expectCenter = {{0.33333, 0.33333},{0.66667, 0.66667}};
12    // floatArrayCompare returns true if the difference between elements
13    // on the same position of the compared containers in absolute value are lower
14    // than prescribed treshold (by default 1e-5)
15    EXPECT_TRUE(floatArrayCompare((centers.getDataByDim<2>()), expectCenter));
16
17    // Test of the mesh connections between various dimensions (see Section 2.4.2)
18    std::vector<std::vector<size_t>> expCon20 = { { 0, 1, 2 }, { 1, 2, 3 } };
19    std::vector<std::vector<size_t>> expCon21 = { { 0, 1, 2 }, { 2, 3, 4 } };
20    std::vector<std::vector<size_t>> expCon12 = { { 0 }, { 0 }, { 0, 1 }, { 1 }, { 1 } };
21    std::vector<std::vector<size_t>> expCon02 = { { 0 }, { 0, 1 }, { 0, 1 }, { 1 } };
22
23    EXPECT_EQ((mesh.connections<2,0>().getDataByPos<0>()), expCon20);
24    EXPECT_EQ((mesh.connections<2,1>().getDataByPos<0>()), expCon21);
25    EXPECT_EQ((mesh.connections<1,2>().getDataByPos<0>()), expCon12);
26    EXPECT_EQ((mesh.connections<0,2>().getDataByPos<0>()), expCon02);
27
28    // Test of proper mesh coloring (see Section 2.4.4)
29    // testProperColoring returns true if the given coloring is proper
30    // with respect to the dimensions setup and the mesh
31    EXPECT_TRUE((testProperColoring<1, 0>(mesh, mesh.coloring< 1, 0 >())));
32 }
```

Archive name	Content
<code>CUDA_DBGVAR.tar.gz</code>	an example of implementation of the general logging tool in CUDA
<code>doxygen-doc.tar.gz</code>	the documentation of the GTMesh library generated from its source code using Doxygen
<code>GTMesh.tar.gz</code>	clone of the GTMesh GIT repository [12]
<code>MultiphaseFlow.tar.gz</code>	code calculating the problems presented in Sections 7.2, 7.3. Each 2D and 3D example is in its respective subdirectory. Both the examples can be run by the <code>run.sh</code> script in their directory.
<code>SinglephaseFlow-stack.tar.gz</code>	code calculating the problem presented in Section 7.1. All examples can be run by the shell script corresponding to the particular mesh, e.g., the <code>run-struct-11.sh</code> builds and runs the program for $\mathcal{T}_{s,1}$ (see Section 7.1). The meshes are located in the <code>meshes</code> subdirectory.

Table A.1: Content of the CD appended to the thesis. To build the computational examples (i.e. `MultiphaseFlow` and `SinglephaseFlow-stack`) by the prepared shell scripts, the `qmake` tool provided by Qt [20] is required.