applied sciences

MDPI

Article

GTMesh: A Highly Efficient C++ Template Library for Numerical Schemes on General Topology Meshes

Tomáš Jakubec and Pavel Strachota *

Department of Mathematics, Faculty of Nuclear Science and Physical Engineering, Czech Technical University in Prague, Trojanova 13, 120 00 Praha 2, Czech Republic; jakubec.t@email.cz
* Correspondence: pavel.strachota@fjfi.cvut.cz; Tel.: +420-778-546-112

Abstract: This article introduces GTMesh, an open-source C++ library providing data structures and algorithms that facilitate the development of numerical schemes on general polytopal meshes. After discussing the features and limitations of the existing open-source alternatives, we focus on the theoretical description of geometry and the topology of conforming polytopal meshes in an arbitrary-dimensional space, using elements from graph theory. The data structure for mesh representation is explained. The main part of the article focuses on the implementation of data structures and algorithms (computation of measures, centers, normals, cell coloring) by using State-of-the-Art template metaprogramming techniques for maximum performance. The geometrical algorithms are designed to be valid regardless of the dimension of the underlying space. As an integral part of the library, a template implementation of class reflection in C++ has been created, which is sufficiently versatile and suitable for the development of numerical and data I/O algorithms working with generic data types. Finally, the use of GTMesh is demonstrated on a simple example of solving the heat equation by the finite volume method.

Keywords: polyhedral meshes; mesh topology; numerical library; template metaprogramming; class reflection in C++; numerical schemes; finite volume method

Citation: Jakubec, T.; Strachota, P. GTMesh: A Highly Efficient C++ Template Library for Numerical Schemes on General Topology Meshes. *Appl. Sci.* **2023**, *13*, 8748. https://doi.org/10.3390/app13158748

Academic Editors: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 13 May 2023
Revised: 13 July 2023
Accepted: 24 July 2023
Published: 28 July 2023

1. Introduction

Numerical algorithms for the solution of problems for partial differential equations often rely on meshes covering the computational domain. The prominent representatives of mesh-based methods are the finite element method (FEM) [1–4] and the finite volume method (FVM) [5–7]. In order to tackle domains with nontrivial geometries, efficient tools for unstructured mesh representation and manipulation are needed. In particular, using general polygonal or polyhedral meshes comprising a relatively small number of highly complex cells can be beneficial in several situations.

Using FVM to solve problems involving, e.g., reacting multiphase flows [8,9], the number of necessary evaluations of computationally costly terms in the governing equations can be reduced by using polyhedral meshes. In the CFD domain, the properties of polyhedral meshes and their possible advantages, in terms of convergence rate and computational costs, have recently started to be investigated systematically [10–13], and their support is available in popular packages, such as ANSYS Fluent, OpenFOAM [14] or AVL FIRE. Efficient finite volume schemes, specifically tailored to polyhedral meshes, are in active development [15,16].

FEM traditionally uses tetrahedral or hexahedral meshes (in 3D). However, FEM has recently been successfully generalized to meshes of convex polyhedrons, which has brought about significant advantages [17–20]. Their applications also extend to computer graphics [21].

For developers of numerical codes, several open-source projects offer mesh handling support, but none of them can provide the complete set of features required by high-performance parallel FVM and FEM codes. Notable examples are outlined below:

- OpenMesh [22] is aimed at applications in computer graphics, and provides tools for mesh modification. However, it only supports polygonal meshes for the representation of surfaces in 3D.
- PUMI (Parallel Unstructured Mesh Infrastructure) [23] is a complex library supporting distributed mesh storage and processing via MPI [24]. By default, the mesh representation by means of the MDS (Mesh Data Structure) submodule only accepts a limited set of topological types known a priori at compile time, in order to achieve high performance.
- ViennaGrid [25] is a modern library leveraging the concepts of template metaprogramming and iterators available in C++. It provides data structures for arbitrary-dimensional meshes and also a number of specialized data types and mesh algorithms that are often limited to certain dimensions or mesh types.
- MOAB (Mesh-Oriented datABase) [26] is an extensive library that allows general topology meshes and supports mesh refinement, decomposition, parallel I/O and other features. Despite being written in C++, it does not take advantage of templates, and its performance has been shown to be relatively poor when processing polyhedral meshes [27]. MOAB always represents mesh geometry in a 3D coordinate system.
- DUNE (Distributed and Unified Numerics Environment) [28] is a framework for numerical computations primarily (but not exclusively) aimed at FEM. It is a mature and very complex project with incomplete documentation, which also supports general topology meshes.
- TNL (Template Numerical Library) [29] is a dynamically evolving framework for implementing efficient numerical algorithms in C++, taking advantage of State-of-the-Art C++ programming paradigms and exposing both CPU and GPU programming through a unified interface. Recently, support for general topology polytopal meshes [27] has been added.

With all the above in mind, we introduce GTMesh, a library created to facilitate rapid development of numerical schemes and data postprocessing algorithms on general polytopal meshes in an arbitrary dimension. GTMesh is implemented as a modern C++ header-only library (compliant with the C++14 standard [30]). GTMesh makes extensive use of template metaprogramming, together with advanced concepts, such as SFINAE [31], on account of which it can provide generic data structures and mesh algorithms that compile for the desired use case and provide maximum run time performance. Using GTMesh, it is possible to write the dimension-agnostic code of a complete numerical solver, including the numerical scheme, the data association with the mesh and the data I/O. The dimension of the problem is then specified as a template parameter at a single point in the code. Currently, GTMesh is a relatively small open-source project, which is publicly available, together with introductory documentation (see Data Availability Statement at the end).

This paper serves as an introduction to the concepts used in GTMesh. In Section 2, the theoretical description of general topology meshes in $\mathbb{R}^d$, $d \in \mathbb{N}$, is laid out. Section 3 presents the general data structure for the mesh representation used in this project. The implementation details are explained in Section 4, including the data structures (Sections 4.1 and 4.2) and the mesh algorithms (Section 4.3). In this section, the ideas are often presented in the context of FVM schemes. For simplicity, geometrical considerations are demonstrated on 2D polygons or 3D polyhedra, despite the possibility of generalization into $\mathbb{R}^d$, $d > 3$. Section 5 is devoted to a useful concept of class reflection, which finds utility in generic numerical and data I/O algorithms. Its implementation is based on original ideas combining C++ class templates and preprocessor macros, in order to create a transparent and comfortable interface for the user. Finally, a simple example program that solves heat conduction in 3D, by the FVM scheme, is demonstrated in Section 6.

2. Geometry and Topology of Unstructured Meshes

The following definitions are motivated by meshing computational domains for the purpose of finite volume schemes. For an arbitrary set, $S \subset \mathbb{R}^d$, denoted by $m(S)$, the d-dimensional Lebesgue measure of S, and by $\tilde{m}(S)$, the $(d-1)$-dimensional Hausdorff measure of S.

Definition 1. *Let the spatial domain Ω be a bounded polytope in $\mathbb{R}^d$, $d \in \mathbb{N}$. Let $\mathscr{T}$ be a set of polytopal cells, and denote by $\mathscr{E}$ the set of all faces that constitute the boundaries of all elements of $\mathscr{T}$. $\mathscr{T}$ is called a d-dimensional conforming mesh on Ω, if the following properties are satisfied:*

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

Next, we introduce the notation for the sets of *elements* of different dimensions that form the geometry of the mesh. For example, in a 3D mesh, the boundaries of the 3D cells consist of 2D faces, which in turn represent flat surfaces bounded by 1D edges, which in turn span between their two 0D vertices. In order to avoid confusion with the notion of element in FEM, these objects are also called *entities* [27].

Definition 2. *Let $\mathscr{T}$ be a d-dimensional mesh, where $d \in \mathbb{N}$. The set of elements of dimension $k \in \{0, 1, \ldots, d\}$ is denoted by $\mathscr{T}^k$. $N^k_{\mathscr{T}} = \left|\mathscr{T}^k\right|$ is the number of elements of dimension k in $\mathscr{T}$. Finally, the complete system of geometrical elements is defined as*

$$\mathscr{T}^* = \bigcup_{k=0}^{d} \mathscr{T}^k.$$

By Definition 2, we have $\mathscr{E} = \mathscr{T}^{d-1}$. In terms of the mutual inclusion of elements of various dimensions, we define the relation of connection on $\mathscr{T}^*$:

Definition 3. *$e, f \in \mathscr{T}^*$ are connected* $\iff \left(e \subset \bar{f} \vee f \subset \bar{e}\right)$.

An example of mesh connections can be seen in Figure 1.

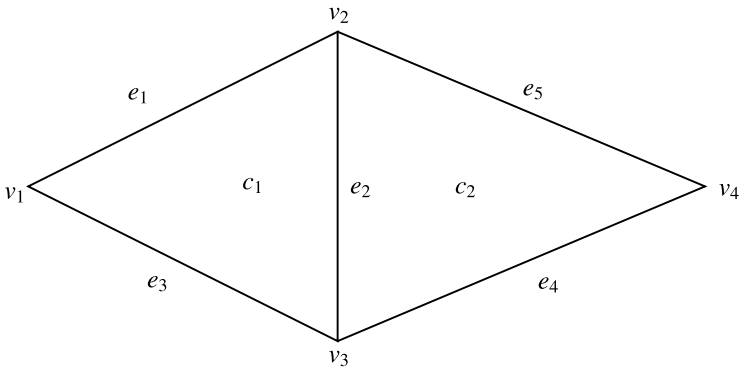


Figure 1. Example of connections in a simple 2D mesh. Vertices connected to element c_1 are $\{v_3, v_2, v_4\}$. The cells connected to element v_1 are $\{c_1, c_2\}$.

In a similar intuitive way, the neighborhood of elements is defined:

Definition 4. *Let $e \in \mathscr{T}^*$. Then, the neighborhood of e is defined as*

$$N(e) = \left\{e' \in \mathscr{T}^* \middle| e' \text{ is connected to } e\right\}.$$

Moreover, we denote the subset of the connected elements with the given dimension $k \in \{0, 1, \ldots, d\}$ as

$$N^k(e) = N(e) \cap \mathscr{T}^k.$$

Graph Description of a General Topology Mesh

All connections in the mesh can be represented by means of a graph $G_{\mathscr{T}^*} = (V_{\mathscr{T}^*}, E_{\mathscr{T}^*})$, where the vertices of the graph match the elements of the mesh, and where the edges of the graph correspond to the connections between them. The vertices $V_{\mathscr{T}^*}$ of $G_{\mathscr{T}^*}$ are grouped into layers by dimensions of elements:

$$V_{\mathscr{T}^*} \cong \mathscr{T}^* = \mathscr{T}^0 \cup \mathscr{T}^1 \cup \ldots \cup \mathscr{T}^d; \tag{1}$$

$$V^k_{\mathscr{T}^*} \cong \mathscr{T}^k. \tag{2}$$

The edges of $G_{\mathscr{T}^*}$ are defined as

$$E_{\mathscr{T}^*} = \left\{ (e, e') \in V^2_{\mathscr{T}^*} \middle| e, e' \text{ are connected} \right\}. \tag{3}$$

For an example, see Figure 2.

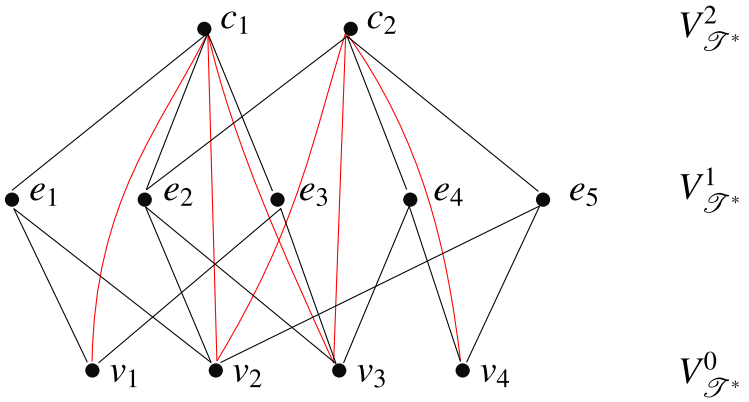


Figure 2. Example of the graph $G_{\mathscr{T}^*}$ associated with the mesh shown in Figure 1. The edges in the graph are presented by black or red lines. The edges between cells and vertices are highlighted in red.

The graph $G_{\mathscr{T}^*}$ contains all information about the topology of the mesh. As the connection of elements is a symmetrical relation, the graph $G_{\mathscr{T}^*}$ is de facto undirected, i.e., $(e, e') \in E_{\mathscr{T}^*} \iff (e', e) \in E_{\mathscr{T}^*}, \forall e, e' \in V_{\mathscr{T}^*}$. Additionally, for a simpler description of the connections, i.e., of the graph edges, we denote a subset of graph edges from dimension d_1 to dimension d_2 as

$$E^{d_1,d_2}_{\mathscr{T}^*} = \left\{ (e, e') \in E_{\mathscr{T}^*} \middle| e \in V^{d_1}_{\mathscr{T}^*}, e' \in V^{d_2}_{\mathscr{T}^*} \right\}. \tag{4}$$

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*

$$[\mathbb{A}_G]_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E, \\ 0 \text{ if } (v_i, v_j) \notin E, \end{cases} \tag{5}$$

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 the idea of a connection matrix, closely related to the adjacency matrix. The connection matrix from dimension d_1 to dimension d_2 reads

$$\left[\mathbb{A}^{d_1,d_2}_{G_{\mathscr{T}^*}}\right]_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E,\ v_i \in V^{d_1}_{\mathscr{T}^*},\ v_j \in V^{d_2}_{\mathscr{T}^*}, \\ 0 \text{ if } (v_i, v_j) \notin E,\ v_i \in V^{d_1}_{\mathscr{T}^*},\ v_j \in V^{d_2}_{\mathscr{T}^*}, \end{cases} \tag{6}$$

where $\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_2} \in \mathbb{R}^{N_{\mathscr{T}}^{d_1} \times N_{\mathscr{T}}^{d_2}}$. The connection matrix is a rectangular block of $A_{G_{\mathscr{T}^*}}$.

The primary aim of the data structure representing any unstructured mesh is to store all connections of the elements in the mesh. Equivalently, it must contain enough information for the reconstruction of the whole $G_{\mathscr{T}^*}$. The graph is fully described by its adjacency matrix. Due to certain properties of the adjacency matrix of $G_{\mathscr{T}^*}$, it is possible to reduce the amount of connections stored. First, it holds that

$$\left(\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_2}\right)^T = \left(\mathbb{A}_{G_{\mathscr{T}^*}}^{d_2,d_1}\right), \tag{7}$$

where the $\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_2} \in \{0,1\}^{\left|V_{\mathscr{T}^*}^{d_1}\right| \times \left|V_{\mathscr{T}^*}^{d_2}\right|}$ is a rectangular submatrix of $\mathbb{A}_{G_{\mathscr{T}^*}}$ reflecting the connections from $\mathscr{T}^{d_1}$ to $\mathscr{T}^{d_2}$. The second property is the dependence between adjacency matrices between specified dimensions. The relation reads

$$\begin{aligned}\left[\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_2}\right]_{ij} &= \operatorname{connect}\left(\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_3}, \mathbb{A}_{G_{\mathscr{T}^*}}^{d_3,d_2}\right) \\ &= \begin{cases} 1 & \text{if } \left(\exists k \in \left\{1,2,\ldots,N_{\mathscr{T}^*}^{d_3}\right\}\right)\left(\left[\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_3}\right]_{ik}\left[\mathbb{A}_{G_{\mathscr{T}^*}}^{d_3,d_2}\right]_{kj} = 1\right), \\ 0 & \text{else,} \end{cases}\end{aligned} \tag{8}$$

where the dimensions d_1, d_2, d_3 satisfy $(d_1 > d_3 > d_2) \vee (d_1 < d_3 < d_2)$. In other words, the Formula (8) represents the chaining of the connections in the graph. 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.

3. Abstract Data Structure for Mesh Representation in $\mathbb{R}^d$

This section describes the system of connections used in this work for storing an unstructured mesh with general topology and dimensions.

According to (7) and (8), it is necessary to store either connections from $V_{\mathscr{T}^*}^k$ to $V_{\mathscr{T}^*}^{k-1}$ or from $V_{\mathscr{T}^*}^{k-1}$ to $V_{\mathscr{T}^*}^k$, because those connections cannot be correctly obtained otherwise. Therefore, we introduce a basic data structure as a sub-system of connections suitable to represent any unstructured mesh in any dimension. Let us first consider the case $d = 3$. The chosen data structure in 3D stores the connections

$$E_{\mathscr{T}^*}^* = E_{\mathscr{T}^*}^{*3,2} \cup E_{\mathscr{T}^*}^{*2,2} \cup E_{\mathscr{T}^*}^{2,3} \cup E_{\mathscr{T}^*}^{2,1} \cup E_{\mathscr{T}^*}^{1,0}, \tag{9}$$

where the $E_{\mathscr{T}^*}^{*3,2}$ and $E_{\mathscr{T}^*}^{*2,2}$ are auxiliary connections enabling direct iteration over cell boundaries. The $E_{\mathscr{T}^*}^{*3,2}$ are pointers from each cell to one of its faces, while the references $E_{\mathscr{T}^*}^{*2,2}$ are between the faces, and they connect the faces making up the boundary of each single cell. Other connections are defined according to Equation (4). For better understanding of the connections between the elements, see Figure 3 presenting connections in a 3D mesh.

This concept is extensible to any dimension. The formula describing the system of connections stored in d-dimensional unstructured mesh is

$$E_{\mathscr{T}^*}^* = E_{\mathscr{T}^*}^{*d,d-1} \cup E_{\mathscr{T}^*}^{*d-1,d-1} \cup E_{\mathscr{T}^*}^{d-1,d} \cup E_{\mathscr{T}^*}^{d-1,d-2} \cup \ldots \cup E_{\mathscr{T}^*}^{2,1} \cup E_{\mathscr{T}^*}^{1,0}. \tag{10}$$

The system of connections $E_{\mathscr{T}^*}^*$ is presented in Figure 4.

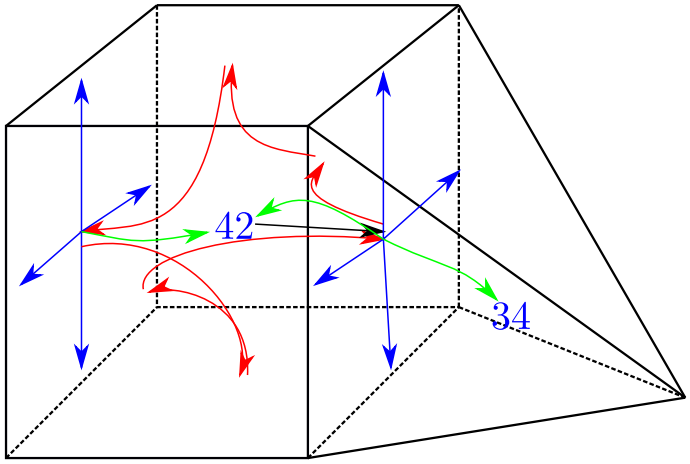
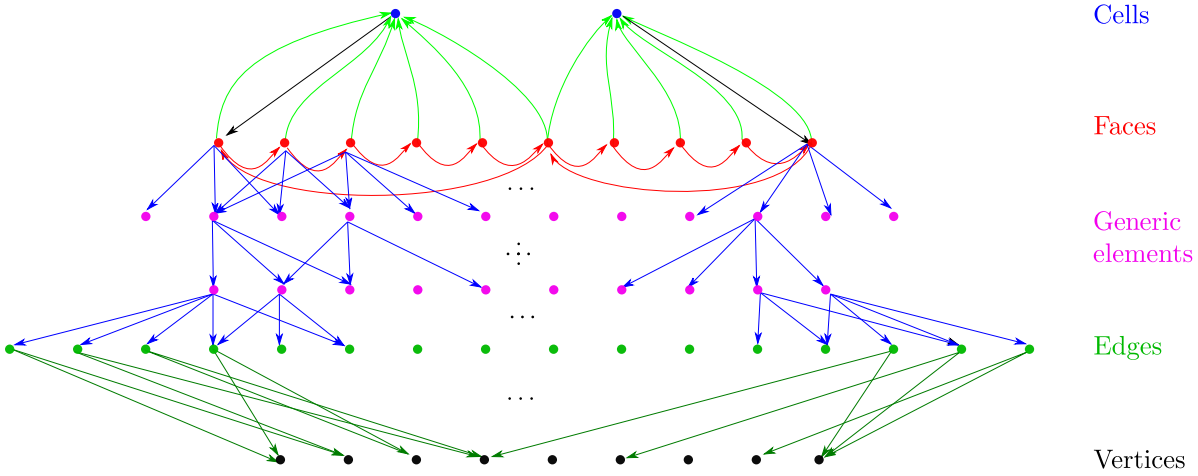


Figure 3. Connections on an example 3D mesh corresponding to the graph presented in Figure 4.



Summary of connections

Arrow	Connection subset	From	To	#connections *from* element
←	$E^{*d,d-1}_{\mathcal{T}^*}$	cells	faces	1
←	$E^{*d-1,d}_{\mathcal{T}^*}$	faces	cells	max. 2
←	$E^{*d-1,d-1}_{\mathcal{T}^*}$	faces	faces	1
←	$E^{*k,k-1}_{\mathcal{T}^*}$	generic dim. k	generic dim. $k-1$	arbitrary
←	$E^{*1,0}_{\mathcal{T}^*}$	edges	vertices	2

Figure 4. An example of a graph representing the topology of a generic d-dimensional mesh. If $d > 3$, one or more violet layers of generic elements with dimensions $k \in \{2, 3, \ldots, d-2\}$ appear. Each of these generic elements refers to the elements of its boundary, using blue arrows $E^{k,k-1}_{\mathcal{T}^*}$. In the last column of the table, the numbers of connections of the given type from each element are reported.

4. The GTMesh Library

The GTMesh library is a C++ project designed for efficient work with polytopal meshes. In order to achieve both user friendliness and computational efficiency, it utilizes modern C++ paradigms. The architecture of GTMesh aims at maintainability and extensibility, using, e.g., the open–closed principle [32].

From the construction point of view, the internal structure of the mesh storing the chosen system of connections (10) is very similar to formats of sparse matrices. However, the structures representing elements are designed to be more convenient for the user.

The library provides commonly used mesh algorithms determining, e.g., connections between elements of any desired dimensions, neighborhood of elements, measures of all elements of a mesh of an arbitrary dimension and normals to cell faces.

The architecture of GTMesh is split into four main parts:

- storing the mesh topology;
- associating data with the mesh;
- calculating various properties of the mesh;
- exporting and importing mesh-related data.

The construction parts of GTMesh are described in the following sections. Section 4.1 discusses the structure storing the mesh topology. Section 4.2 presents the system of data mapping to the mesh. This data mapping system significantly contributes to the user friendliness of GTMesh. Next, the auxiliary functions for calculating, e.g., elements centers and measures are described in Section 4.3. Section 5 describes a unique system of class reflection, developed to automate basic operations with C++ structures or classes such as serialization, deserialization or arithmetical operations. This concept improves the development efficiency, as it reduces the amount of routine work on data interface coding.

4.1. Mesh Data Structure

The complete C++ data structure storing a mesh consists of arrays containing the mesh elements. The representation of a mesh element is to be understood as a simple C++ data structure containing the respective references from (10). An example for the dimension $d = 3$ is depicted in Figure 5. In addition, the structures for cells and faces contain two auxiliary data elements: *center* (see Section 4.3) and *flag*.

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

Figure 5. Scheme of the chosen representation demonstrated on a 3D unstructured mesh.

The elements are collected in a data structure called `MeshElements`, which is de facto the unstructured mesh itself. Due to the template implementation of the mesh element data structure, it is possible to automatically generate the system of elements based on the template arguments, i.e., only one definition of the `MeshElements` class is required, e.g., for definitions of 2D or 3D mesh. The dimension of the mesh is given by a template parameter.

Additionally, it is possible to prescribe the maximum number of sub-elements of faces and other elements which have references represented by blue arrows in Figure 4. This number is called `Reserve`. When `Reserve` is defined, it is possible to embed the references directly into the corresponding mesh element data structure, avoiding dynamic memory allocation. For example, when the number of sub-elements of faces in a 3D mesh is prescribed to three, the stored mesh can only have triangular faces. Note that, due to the chosen representation, the number of faces of one cell is unlimited in any dimension.

4.2. Associating Data with the Mesh

Any numerical method for solving a system of partial differential equations 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 with mesh elements is specific to their dimension. For example, finite volume methods use solution values for the computationally significant elements, i.e., cell centers, cell faces or both. However, additional data storage may be allocated, for each vertex to hold the results of intermediate calculations.

GTMesh provides a data container, named `MeshDataContainer`, to conveniently store data associated with mesh elements. Once the mesh geometry and topology has been provided, `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 $\mathtt{T}_1$, $\mathtt{T}_2$, ..., $\mathtt{T}_n$ associated with all mesh elements of dimensions $d_1, d_2, \ldots, d_n$, respectively. The dimensions specifiers $(d_1, d_2, \ldots, d_n)$ 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 interfaces similar to `std::vector<`$\mathtt{T}_1$`>` ... `std::vector<`$\mathtt{T}_n$`>`. For each $i \in \{1, \ldots, n\}$, 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 $(d_1, d_2, \ldots, d_n)$;
2. By dimension d. In this case, the ith vector is returned, where i is the first integer in the sequence $(1, 2, \ldots, n)$, 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 Listing 1). 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 element dimension and index, present in every `MeshElement` data structure.

Listing 1. Presentation of the usage of the `MeshDataContainer` class. The data container allocates four different data types to dimensions of the mesh. Upon construction, the `MeshDataContainer` class allocates data according to the dimensions of the provided mesh. Then, the example presents the possibility of accessing data, using elements of the mesh or the underlying data collections, by explicitly specifying the dimension or position template parameter.

```
UnstructuredMesh<3, double> mesh3d;
mesh3d.load("mesh_file.vtk")
// Associate data to a mesh
// automatically allocates data according to the mesh dimensions.
// Type char -> cells, int and double -> faces and float -> vertices.
MeshDataContainer<std::tuple<char, int, double, float>, 3,2,2,0> meshData(mesh3d);

// Accesses the value corresponding to the first vertex.
meshData[mesh3d.getVertices()[0]];
// equivalent to
// getDataByDim returns the first collection corresponding to the desired dimension.
meshData.getDataByDim<0>()[0];

// Redundant mappings to a dimension are accessible via getDataByPos only.
meshData.getDataByPos<2>();
```

4.3. Algorithms

This section presents mesh algorithms provided by the GTMesh library. Each algorithm is implemented in a member function of a separate class template. The respective member function accepts a reference to a `MeshElements` instance as a parameter: thus, the set of functions may be extended without modifying the `MeshElements` structure. Therefore, the data structures presented in Section 4.1 do not have any calculation methods, in contrast to, e.g., OpenFOAM [14]. Generally, each external function aims to calculate the respective property for the whole mesh at once, e.g., calculating measures of all elements (of all dimensions) in the mesh. Note that function templates cannot generally be used for this purpose instead of class templates, as C++ does not support partial specialization of function templates. Whenever possible, the functionality of the respective class is wrapped in a function template, to simplify the user interface.

4.3.1. Iteration Over Mesh Structure

The first realized algorithm, `MeshApply::apply()`, provides a unified approach to iterating the connections in the mesh (i.e., `MeshElements`) according to Figure 4: for example, it is able to perform an operation for each vertex (target element dimension) connected to a cell (source element dimension), which would require three nested loops. The dimensions

of the source and target elements are prescribed by template parameters `StartDim` and `TargetDim`. The operation to be performed is provided by a reference to a callable object (e.g., lambda function, function pointer).

This functionality guarantees that the desired operation will be performed with the indices of each element of the source element dimension and all the elements of the target element dimension connected to it. However, it is not guaranteed that the function will be called exactly once for each pair of elements: if this is a concern, multiple calls have to be handled by the user-defined algorithm itself.

Due to the symmetry of the connection relation (Definition 3), the iterations where the target element dimension is higher than the source element dimension (e.g., loop over cells connected to vertices) can be realized by looping with swapped dimensions and by only providing the correct indices of the target and source elements. Hence, GTMesh is able to realize such loops, even though the connections from lower-dimension elements to the ones with higher dimension are not stored in memory.

4.3.2. Determining the Elements' Connections

By means of `MeshConnections::connections()`, the connection matrix $\mathbb{A}^{d_1,d_2}_{G_{\mathscr{T}^*}}$ is calculated. The dimensions d_1, d_2 are given by the template parameters `StartDim` and `TargetDim`. The result is a `MeshDataContainer` mapping a vector of indices of the connected elements of the `TargetDim` dimension for each element of the `StartDim` dimension. For an example of using `MeshConnections`, see Listing 2. `MeshConnections` provides the functionality of the map "connect" introduced in Equation (8).

Listing 2. An example of using `MeshConnections`. The `MeshConnections::connections()` member function 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 connections can optionally be returned in ascending order.

```
// Connected vertices to the cells
auto conCellToVert = MeshConnections<3,0>::connections(mesh);
for (auto& cell : mesh.getCells()){
        conCellToVert[cell]; // Vector of connected vertices to the given cell
}

// Connections in the original order in the mesh
auto conCellToVertOrig = MeshConnections<3,0,Order::ORDER_ORIGINAL>::connections(
    ↪ mesh);
for (auto& cell : mesh.getCells()){
        conCellToVertOrig[cell]; // Vector of vertices connected to the given cell
}

// Detection of the cells connected to a vertex
auto conVertToCell = MeshConnections<0,3>::connections(mesh);
for (auto& vert : mesh.getVertices()){
        conVertToCell[vert]; // Cells connected to the given vertex
}
```

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

4.3.3. Determining the Elements' Neighborhood

`MeshNeighborhood::neighbors()` determines the neighborhood of elements. In terms of graph theory, the neighborhood of a vertex v is a set of graph vertices that are connected to 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 $e \in V^d_{\mathscr{T}^*}$, the set of neighboring elements of dimension d_2 connected by elements of dimension d_1 reads

$$N^{d_1,d_2}_{G_{\mathscr{T}^*}}(e) = \left\{ f \in V^{d_2}_{\mathscr{T}^*} \middle| \left(\exists g \in V^{d_1}_{\mathscr{T}^*}\right)((g,f) \in E_{\mathscr{T}^*} \wedge (g,e) \in E_{\mathscr{T}^*} \wedge e \neq f) \right\}, \tag{11}$$

where $d, d_1, d_2 \in \{0, 1, \ldots, d_{\mathscr{T}}\}$. In the `MeshNeighborhood` class, the parameters d, d_1, d_2 are named `StartDim`, `ConnectingDim` and `ConnectedDim`, respectively. This algorithm returns a vector of indices of neighboring elements according to (11) for each element of dimension d. If a broader neighborhood is needed, the neighborhoods of neighboring elements can be combined, using the `std::set_union()` function.

4.3.4. Calculation of Proper Coloring of a Mesh

The last of the algorithms related to graph representation of the mesh topology is the coloring algorithm `ColorMesh::color()`. The problem consists of the proper coloring of the elements of dimension d according to the connections of dimension d_1 [33]. This algorithm can be used to advantage, to prevent race conditions during multi-threaded parallel computation.

The template parameters of the `ColorMesh` class are as follows: d as `ColoredDim`; d_1 as `ConnectingDim`; and `Method`, to select one of the two supported coloring methods. The greedy method utilizes the first free color index when determining the color index for an element. Obviously, this approach may lead to uneven distribution of color indices in the mesh. If this turns out to be a limiting factor for parallel performance, random update strategy can be used instead: first, a proper coloring is obtained, using the greedy algorithm; then, a second phase is launched, which rebalances the color indices randomly, while still maintaining the proper coloring in each step.

4.3.5. Calculation of Element Centers

The previous algorithms work with the mesh as a topological object, i.e., they do not use the coordinates of vertices. Now, we will describe the algorithms that calculate some significant geometrical properties of the mesh. We begin with an algorithm calculating the center points of all objects in the mesh, which is crucial for implementing cell-centered FVM schemes.

The center of each element is calculated as an average of the positions of the centers of all the connected sub-elements. For example, the center of a cell is an average of the centers of the cell's faces. The advantage of this approach is the reduction of the depth of iteration over the mesh (see the example of the algorithm in Figure 6). Because the algorithm uses the previously calculated values, the computation may 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 $d_{\mathscr{T}} = 2$, it is sufficient to visit the sub-elements of dimension 1. The scheme of the algorithm is as follows:

1. set the dimension $d = 1$;
2. for all elements $e \in \mathscr{T}^d$, calculate the center of e as $\boldsymbol{x}_e = \frac{1}{|N^{d-1}(e)|} \sum_{f \in N^{d-1}(e)} \boldsymbol{x}_f$ by a loop over sub-elements of e;
3. if $d < d_{\mathscr{T}}$ 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 results of both algorithms may differ if there are faces with different numbers of vertices. The `computeCenters()` function calculates the centers of all elements in the mesh, using the algorithm presented above.

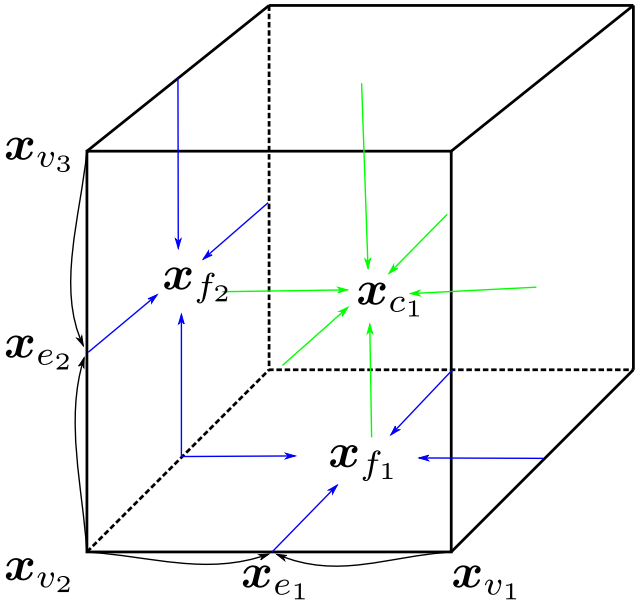


Figure 6. An example of the algorithm for calculating the element centers in a simple 3D mesh. The center point of an element is calculated as an average of the center points of its sub-elements. For example, the cell center point denoted $\boldsymbol{x}_c$ is the average of the centers of all connected faces, denoted $\boldsymbol{x}_{f_i}$.

4.3.6. Calculation of Element Measures

The next algorithm calculates the Hausdorff measures of elements, with respect to their dimension $1 < d \leq d_{\mathscr{T}}$, where $d_{\mathscr{T}}$ is the dimension of the mesh. We assume that every polytope $e \in \mathscr{T}^d$ is a star domain, with respect to its center $\boldsymbol{x}_e$ [34]. For $d > 1$, such polytopes can be subdivided into pyramids $P_{e,e'}$ with (planar) bases formed by their sub-elements $e' \in \mathscr{T}^{d-1}$ and a common top $\boldsymbol{x}_e$. An example of subdivision of an element into pyramids is presented in Figure 7. This can be expressed as

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

(see Definition 4). The d-dimensional Hausdorff measure of e is then given by

$$m(e) = \sum_{e' \in N^{d-1}(e)} m\left(P_{e,e'}\right). \tag{12}$$

Let us denote by $\boldsymbol{v}$ the geometrical position of each vertex $v \in \mathscr{T}^0$. Then, the measure of the pyramid $P_{e,e'}$ reads

$$m\left(P_{e,e'}\right) = \frac{1}{d} m\left(e'\right) \operatorname{dist}(V_{e'}, \boldsymbol{x}_e), \tag{13}$$

where $V_{e'}$ is a linear manifold of dimension $d-1$, containing the element e', and $m(e')$ is the $(d-1)$-dimensional measure of the pyramid base, calculated recursively by the same algorithm. To find the height of the pyramid $\operatorname{dist}(V_{e'}, \boldsymbol{x}_e)$, the Gram–Schmidt process is applied to the system of linearly independent vectors:

$$\left(\boldsymbol{v}_{e',2} - \boldsymbol{v}_{e',1}, \boldsymbol{v}_{e',3} - \boldsymbol{v}_{e',1}, \ldots, \boldsymbol{v}_{e',d} - \boldsymbol{v}_{e',1}, \boldsymbol{x}_e - \boldsymbol{v}_{e',1}\right), \tag{14}$$

where $\boldsymbol{v}_{e',i} \in \mathscr{T}^0 \cap e'$ are unique vertices of e'. Note that an element of dimension $d-1$ has at least d vertices. This results in an orthogonal system:

$$\left(\boldsymbol{y}_2, \boldsymbol{y}_3, \ldots, \boldsymbol{y}_d, \tilde{\boldsymbol{y}}\right), \tag{15}$$

where the last vector, $\tilde{\boldsymbol{y}}$, is not normalized. Finally, we calculate

$$\operatorname{dist}(V_{e'}, \boldsymbol{x}_e) = |\tilde{\boldsymbol{y}}|.$$

For $d = 1$, the 1D measure (length) of an edge $e \in \mathscr{T}^1$ is calculated by

$$m(e) = |\boldsymbol{v}_A - \boldsymbol{v}_B|, \tag{16}$$

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

The above algorithm is implemented by the `computeMeasures()` function, and it returns a `MeshDataContainer` with measures of all elements in the mesh (except vertices). The `computeMeasures()` function calculates the measures from lower dimensions to higher ones, and it utilizes, to advantage, the already-calculated measures of lower-dimensional elements. This function also supports the compensation for non-planar elements [35].

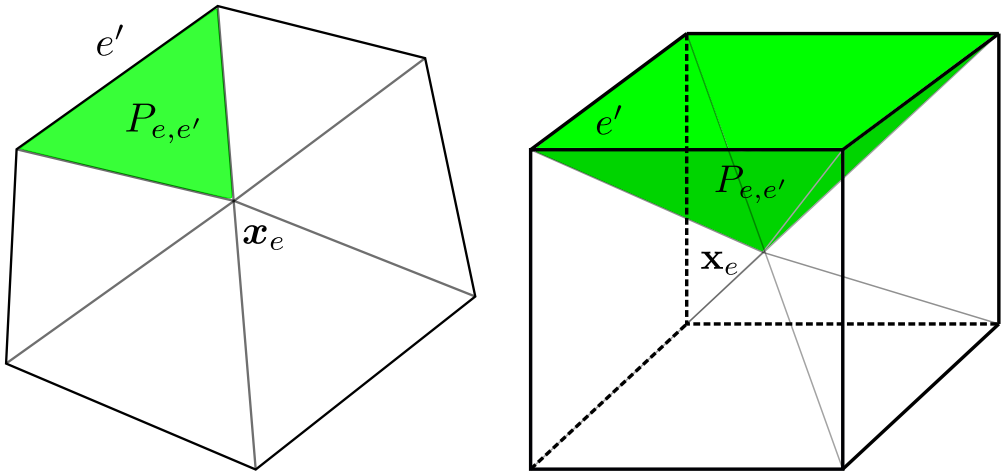


Figure 7. An example of the computation of a 2D Hausdorff measure of a polygonal element, $e \in \mathscr{T}^2$, and of a 3D measure of a polyhedron, $e \in \mathscr{T}^3$.

4.3.7. Calculation of Face Normal Vectors

Another common operation is to calculate the outward-pointing normal vector $\boldsymbol{n}_{e'}$ to each face $e' \in N^{d_{\mathscr{T}}-1}(e)$ of each cell $e \in \mathscr{T}^{d_{\mathscr{T}}}$. After constructing the system of vectors (14) and applying the Gram–Schmidt process, to obtain (14), the normal vector is calculated as

$$\boldsymbol{n}_{e'} = -\frac{\tilde{\boldsymbol{y}}}{|\tilde{\boldsymbol{y}}|}.$$

The function responsible for the calculation of normal vectors is `computeFaceNormals()`. This function calculates a normal vector for each face in the mesh. The calculated normal vector points from the right cell (`CellRightIndex`) to the left cell (`CellLeftIndex`) are given by the data structure.

4.3.8. Import and Export of the Mesh

GTMesh provides tools for importing and exporting the mesh and the data associated with the mesh. The supported formats are VTK [36] for 2D and 3D meshes and FPMA for 3D meshes. The loading of meshes is realized by the `VTKMeshReader` and `FPMAMeshReader` classes, which accept the dimension of the source mesh as a template parameter. Similarly, there are classes for exporting meshes to the respective formats, i.e., `VTKMeshWriter` and `FPMAMeshWriter`. As the export operation is expensive, both mesh writers support a caching mechanism. If a mesh with the same topology is exported repeatedly via the same writer instance, the previously cached data are used. Note that `VTKMeshWriter` needs to know the types of cells to be exported. This information is not stored in the mesh directly, but it can be obtained from the reader instance that loaded the mesh. If the cell type is identified as a

generic polyhedron, the cell is tessellated into tetrahedrons constructed from cell center, face center and edge vertices.

Support for other popular formats used, e.g., by ANSYS Fluent or CFX can be added in future, by implementing the respective classes. In addition, the readily available conversion tools provided by OpenFOAM [14] have been successfully tested.

4.3.9. Import and Export of Mesh Data

As the FPMA format does not support storing data, the only currently supported mesh data export is to the VTK format. The classes responsible for reading and writing data associated with a mesh are `VTKMeshDataReader` and `VTKMeshDataWriter`. The aim is to export data in a format that can be directly visualized using, e.g., ParaView.

The `VTKMeshDataWriter::writeToStream()` member function accepts a reference to an `std::ostream` to which the data will be appended, `MeshDataContainer`, with data associated with cells and a `VTKMeshWriter` instance utilized to export the mesh, as it contains metadata related to the tessellation of cells. The data must have the I/O Traits defined (see Section 5 below), in order to be exported. If a cell tesselation is performed, the data associated with the original cell is exported for all the resulting tetrahedrons.

The `VTKMeshDataReader::readFromStream()` member function accepts an `std::istream` from which the data will be loaded and a `MeshDataContainer` instance with corresponding data types associated with cells in which to store the data. The data types must have I/O Traits defined, in order to be loaded. The data array is allocated according to the length of the loaded array, regardless of the size of the actual mesh.

GTMesh provides other tools for data analysis, especially the `DBGVAR_JSON` debugging macro defined in `debug.h`. This macro prints JSON-formatted logs into a `.json` file. This file can subsequently be analyzed by a number of readily available tools, e.g., using the features and scientific packages in the Python ecosystem.

Finally, GTMesh provides the `BinarySerializer` class to save/load raw data in binary form without any information loss due to rounding. This is useful for creating snapshots of a numerical simulation that can serve as starting points for later continued computation. The endianness is not guaranteed, and depends on the architecture of the host system.

4.4. The `UnstructuredMesh` Wrapper Class

As described in the project architecture (Section 4), GTMesh aims at providing 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 it has no further structure. The only purpose of `UnstructuredMesh` is to provide the algorithms from Sections 4.3 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. An example of `UnstructuredMesh` wrapper usage is shown in Listing 3.

Listing 3. An example of use of the `UnstructuredMesh` wrapper class. This listing illustrates calculating mesh properties, exporting them into a VTK file and loading them back. The exported three quantities associated with the cells are coloring with respect to connections over vertices, center point and inverse measure of cells. The process of calculation of the exported properties illustrates the fundamental design of the function calculating the properties of the `UnstructuredMesh` object.

```
#import <fstream>
#import <GTMesh/UnstructuredMesh/UnstructuredMesh.h>

struct CellData {
    unsigned int color;
    Vertex<3, double> center;
    double invVol;
};
// Create default Traits for the class CellData
MAKE_ATTRIBUTE_TRAIT(CellData, center, color, invVol);

void main() {
    // load mesh
    UnstructuredMesh<3, size_t, double, 6>& mesh;
    auto meshReader = mesh.load("mesh_file.vtk");
    mesh.initializeCenters();

    // Calculate mesh properties and store them into meshData
    MeshDataContainer<CellData, 3> meshData(mesh);
    auto colors = ColorMesh<3,0>::color(mesh);
    auto measures = mesh.computeElementMeasures();
    for(auto& cell : mesh.getCells()){
        meshData[cell].color = colors[cell];
        meshData[cell].center = cell.getCenter();
        meshData[cell].invVol = 1.0 / measures[cell];
    }

    // Export the mesh first
    // In order to enforce the mesh tessellation, prepare the cell types as polyhedrons
    // the correct cell types can be obtained here: meshReader->getCellTypes()
    MeshDataContainer<MeshNativeType<3>::ElementType,3> cellTypes(mesh, MeshNativeType<3>::POLYHEDRON);

    // Write the mesh to file
    VTKMeshWriter<3, size_t, double> writer;
    std::ofstream out3D("mesh.vtk");
    writer.writeHeader(out3D, "test data");
    writer.writeToStream(out3D, mesh, cellTypes);

    // Export the data mapped to the mesh
    VTKMeshDataWriter<3>::writeToStream(out3D, meshData, writer);
    out3D.close();

    // Load the mesh from file
    ifstream in3D("mesh.vtk", std::ios::binary); // VTKMeshDataReader requires binary mode
    VTKMeshReader<3> reader;
    reader.loadFromStream(in3D, mesh);
    mesh.initializeCenters();

    // Read the exported data from the mesh file
    MeshDataContainer<CellData, 3> meshDataIn(mesh);
    VTKMeshDataReader<3, size_t>::readData(in3D, meshDataIn);
    in3D.close();
}
```

5. Class Reflection in C++ Optimized for High-Performance Computing

As part of the GTMesh library, a support tool providing advanced reflection (introspection) of C++ structures and classes has been developed. In general, the aim of this tool is to provide a unified interface for data contained in user-defined structures or classes similar to `std::tuple`. As a result, the data members can be accessed based on an integer index resolved at compile time. As C++14 provides no direct way to introspect objects at run time or compile time, the data access mechanism has to be defined manually by the user. However, GTMesh provides tools that allow to do this in a single line of code.

The data accessed by this mechanism can be arbitrarily related to the data actually stored in the data structure, as demonstrated in Listing 4. Both real and "virtual" data members calculated from the actually stored data have names accessible at run time. In

the context of numerical algorithms, this tool makes it possible to write generic code that implements the following functionalities:

1. Data stored in user-defined data structures can be exported/imported to/from VTK, JSON or binary formats.
2. Arithmetic or other mathematical operations (such as norm calculation) can be performed on the user-defined data structures.

As the above (and possibly other) use cases have different requirements, multiple reflections designed for different purposes can exist for a single class. The iteration over data members is done via static for loops, i.e., resolved at compile time by means of template function recursion. Hence, the generated code is as efficient as manually written code.

At the time of writing, there exist several other open-source projects that aim for data serialization and/or class reflection in C++. A very recent and elegant solution is Cista++ (https://cista.rocks),which implements class reflection with the help of C++17 structured bindings [37]. However, this approach suffers from substantial limitations that prevent using Cista++ with GTMesh: in particular, it does not allow data transformations to create "virtual" data members, and it only works with plain structures without constructors.

5.1. Member Access

A data member denotes any information calculated from the data stored in a data structure. Data members are obtained by their getter and are set by their setter functions. These functions might be arbitrary.

The first part of the architecture is the method for data access. This is realized by the `MemberAccess` template class with several template specializations. This class then provides a unified interface to getting and setting the member data values. `MemberAccess` may be constructed in different ways:

1. Member reference, i.e., `&data_structure::member`, which defines the get, constant get and set operations. These operations are, thus, accessing the member data directly.
2. Pair of getter and setter, where each of these may be a member function or global function (callable object, e.g., lambda function) accepting the data structure instance as a parameter.

5.2. Class Traits

The `MemberAccess` classes are then grouped in the `Traits` class template, which manages their association with the names of the data members. `Traits` has member functions returning the data accessed by `MemberAccess` based on an integer template parameter (see Listing 4).

5.3. Default Traits

In order for the algorithms in GTMesh to be able to access the class reflection functionalities, there has to be a standard mechanism of globally exposing `Traits` for the given user-defined data structures. The solution is based on the `DefaultTraits` class template, which accepts the reflected class as its template parameter. The generic declaration of `DefaultTraits` has no methods or parameters. `Traits` for a particular class are exposed by defining `DefaultTraits` template specialization with a single static member function `getTraits()`, which returns the corresponding `Traits` instance. From `DefaultTraits`, GTMesh derives `DefaultIOTraits` for mesh data I/O and `DefaultArithmeticTraits` for arithmetics (see Listing 5). Both of them inherit the `DefaultTraits` functionality, but a specialization of each template can be provided by the user. The SFINAE paradigm [31] is used to detect whether a particular class has the respective type of `Traits` defined.

Finally, to simplify the definition of default class traits, GTMesh offers several preprocessor macros expanding to the specializations of the above classes. For example, `MAKE_DEFAULT_ATTRIBUTE_TRAITS` expands to `DefaultTraits`, etc., with the expected static member functions and type definitions. As a result, the definition of the default class traits for a data structure requires a single line of code, as demonstrated in Listing 4.

Listing 4. Example of class reflection. There are two types of class traits defined for the `Data` class: the first is generic `DefaultTraits`, which is used as a fallback if a specific traits class is not defined; the other is `DefaultIOTraits`, which provides access to the primary members `density` and `momentum` of the class `Data`, whereas the `DefaulIOTraits` considers the `Data` class to consist of `density` and `velocity`, where `velocity` is calculated upon request from the primary members.

```
class Data {
    double density;
    Vector<3, double> momentum;
    Vector<3, double> getVelocity(){return momentum/density;}
    void setVelocity(const Vector<3, double> velocity){momentum = velocity * density
        ↪ ;}
}

// Macro creates specialization of DefautTraits for Data and names
// according to the attributes names
MAKE_ATTRIBUTE_TRAIT(Data, density, momentum);

// Macro creates specialization of DefaultIOTraits for the Data class
MAKE_CUSTOM_TRAIT_IO(
    Data,
    "density", &Data::density,
    "velocity", std::make_pair(&Data::getVelocity, &Data::setVelocity)
);
// usage ...
// Only values returned as l-value reference can be obtained using getAttr function
DefautTraits<Data>::getTraits().getAttr<0>(qInstance) = 3; //get reference to
    ↪ density
DefautTraits<Data>::getTraits().setValue<1>(qInstance, {3, 6, 9}); // set momentum

DefaultIOTraits<Data>::getTraits().getValue<1>(qInstance); // {1, 2, 3} get velocity
DefaultIOTraits<Data>::getTraits().setValue<1>(qInstance, {1, 1, 1}); // set
    ↪ velocity

DefautTraits<Data>::getTraits().getValue<1>(qInstance); // {3, 3, 3} get momentum
```

Listing 5. Example implementation of operator + for any type with `DefaultArithmeticTraits` or `DefaultTraits` defined, e.g., application of this operator on the data class from Listing 4 would sum up the density and momentum members. This approach allows us to define all common mathematical operations for the classes with traits defined. It is even possible to define maximum value among the members, if the members of the class are comparable.

```
template< typename T1, typename T2 >
struct BinaryPlus
{
   static auto evaluate( const T1& a, const T2& b ) -> decltype( a + b )
   {
      return a + b;
   }
};

template <typename TraitT>
typename std::enable_if<HasDefaultArithmeticTraits<TraitT>::value, TraitT>::type
operator+(const TraitT& op1, const TraitT& op2) noexcept {
    TraitT res;
    // A class performing an operation for each member of the class
    TraitsBinaryExpressionProcessor<BinaryPlus>::evaluate(res, op1, op2);
    return res;
}
```

5.4. Example of Use

By using the concept of class traits, it is possible to generalize complex computational methods to be independent of the data structures utilized by the solved problem, which simplifies their development and debugging. We demonstrate an implementation of a generic version of the fourth-order Runge–Kutta–Merson (RKM) solver, with adaptive time stepping for numerical integration of ordinary differential equation (ODE) systems [38].

Considering the ODE system in the form

$$\dot{\boldsymbol{x}} = f(t, \boldsymbol{x}), \tag{17}$$

where the $\boldsymbol{x} \in \mathbb{R}^N$, $N \in \mathbb{N}$ and $f : \mathscr{J} \times \mathbb{R}^N \to \mathbb{R}^N$, the RKM solver works as presented in Algorithm 1:

Algorithm 1 Pseudo-code of the Runge–Kutta–Merson ODE solver [38].

```
τ = τ_ini ; x^τ = x^τ_ini ; t = T_ini ;
while (|T_ini − t| < |T_ini − T|) {
    if ( |T − t| < |τ| ) {
        τ = T − t ;
    }
    K_1 = f(t, x^τ) ;
    K_2 = f(t + τ/3, x^τ + τ/3 K_1) ;
    K_3 = f(t + τ/3, x^τ + τ/6 (K_1 + K_2)) ;
    K_4 = f(t + τ/2, x^τ + τ/8 (K_1 + 3K_3)) ;
    K_5 = f(t + τ, x^τ + τ(1/2 K_1 − 3/2 K_3 + 2K_4)) ;
    ε = max_element τ/3 |0.2K_1 − 0.9K_3 + 0.8K_4 − 0.1K_5|
    if ( ε < τ ) {
        x^τ = x^τ + τ(1/6 (K_1 + K_5) + 2/3 K_4) ;
        t = t + τ ;
        if (ε == 0) continue ;
    }
    τ = (δ/ε)^0.2 · ωτ ;
}
```

The symbols used there are summarized in Table 1:

Table 1. Symbols used in the Runge–Kutta–Merson pseudo-code (Algorithm 1).

Symbol	Meaning
t	current time level
T	final time
T_{ini}	initial time
τ	time step
τ_{ini}	initial time step
$\boldsymbol{x}^\tau$	numerical solution
$\boldsymbol{x}^\tau_{\text{ini}}$	initial condition for the numerical solution $\boldsymbol{x}^\tau$
δ	tolerance parameter
ω	time step adjustment parameter ($\omega = 0.8$ is recommended)

For more information, see [38,39]. This algorithm is utilized in the example application presented in Section 6 below. Finally, the implementation of the algorithm is illustrated in Listing 6.

Listing 6. Implementation of the Runge–Kutta–Merson [38] algorithm. This algorithm accepts an instance of `Problem` class, which has several properties, such as computational mesh, `ResultType` and mainly `calculateRHS` member function, which accepts `MeshDataContainer` associated with cells of the mesh of the problem and returns $f(t, x)$ as defined in (17). The implementation relies on the fact that the `ResultType` data type has defined (e.g., by the arithmetic traits) arithmetical operations.

```
template <typename Functor, typename ...T, unsigned int Dimension>
void performVectorOperation(Functor&& f, MeshDataContainer<T, Dimension>& ...args) {
    const auto& firstVector = std::get<0>(std::forward_as_tuple(args...));
    for (std::size_t i = 0; i < firstVector.template getDataByPos<0>().size(); ++i) {
        f((args.template getDataByPos<0>()[i])...);
    }
}

template <typename Functor, typename ...T, unsigned int Dimension>
double performVectorReductionMax(Functor&& f, MeshDataContainer<T, Dimension>& ...args) {
    const auto& firstVector = std::get<0>(std::forward_as_tuple(args...));
    double res = std::numeric_limits<double>::lowest();
    for (std::size_t i = 0; i < firstVector.template getDataByPos<0>().size(); ++i) {
        double tmp_res = f((args.template getDataByPos<0>()[i])...);
        if (res < tmp_res) {
            res = tmp_res;
        }
    }
    return res;
}

template <typename Problem, std::enable_if_t<HasDefaultArithmeticTraits<typename Problem::ResultType>::value, bool> =
    ↪ true>
void RKMSolver(Problem& problem,
               MeshDataContainer<typename Problem::ResultType, Problem::MeshType::meshDimension()>& compData,
               double tau_ini, double startTime, double finalT, double delta)
{
    using container_type = MeshDataContainer<typename Problem::ResultType, Problem::MeshType::meshDimension()>;
    container_type Ktemp(compData); //x_ini
    container_type K1(compData), K2(compData), K3(compData), K5(compData), K4(compData);

    double tau = tau_ini, time = startTime;
    while (time < finalT) {
        if (time + tau > finalT) {
            tau = finalT - time;
        }

        problem.calculateRHS(time, compData, K1);
        performVectorOperation(
            [&tau](auto &Ktemp, auto& compData, auto& K1){ Ktemp = compData + (tau * (1.0 / 3.0) * K1); },
            Ktemp, compData, K1
        );

        problem.calculateRHS(time, Ktemp, K2);
        performVectorOperation(
            [&tau](auto &Ktemp, auto& compData, auto& K1, auto& K2){ Ktemp = compData + (tau * (1.0 / 6.0) * (K1 + K2))
                ↪ ; },
            Ktemp, compData, K1, K2
        );

        problem.calculateRHS(time, Ktemp, K3);
        performVectorOperation(
            [&tau](auto &Ktemp, auto& compData, auto& K1, auto& K3){ Ktemp = compData + (tau * (0.125 * K1 + 0.375 * K3
                ↪ )); },
            Ktemp, compData, K1, K3
        );

        problem.calculateRHS(time, Ktemp, K4);
        performVectorOperation(
            [&tau](auto &Ktemp, auto& compData, auto& K1, auto& K3, auto& K4){ Ktemp = compData + (tau * ((0.5 * K1) -
                ↪ (1.5 * K3) + (2.0 * K4))); },
            Ktemp, compData, K1, K3, K4
        );

        problem.calculateRHS(time, Ktemp, K5);
        double error = performVectorReductionMax(
            [](auto& K1, auto& K3, auto& K4, auto& K5)->double{return max(abs(0.2 * K1 - 0.9 * K3 + 0.8 * K4 - 0.1 * K5
                ↪ ));},
            K1, K3, K4, K5
        );
        error *= tau * (1.0 / 3.0);

        if (error < delta) {
            performVectorOperation(
                [&tau](auto& compData, auto& K1, auto& K4, auto& K5){ compData += tau * (1.0 / 6.0) * (((K1 + K5)) +
                    ↪ (4.0 * K4)); },
                compData, K1, K4, K5
            );
            time += tau;
            if (error == 0.0) continue;
        }
        tau *= std::pow(delta/error, 0.2) * 0.8;
    }
}
```

6. Example Application

To introduce GTMesh in action, and to provide a starting point for numerical solver development, an example application has been created (see Data Availability Statement). The numerical solution of the heat equation is demonstrated, using a finite volume scheme for spatial discretization, and using the RKM solver (see Algorithm 1) for temporal discretization. The algorithm works both with 2D and 3D meshes, and is provided in three variants, in terms of parallelization: single-threaded (Section 6.2); OpenMP utilizing graph coloring

to avoid race conditions (Section 6.3); and OpenMP utilizing auxiliary computational data values to avoid race conditions (Section 4.2).

6.1. Problem Formulation and Discretization

Let $\Omega \subset \mathbb{R}^d$, $d \in \{2,3\}$ be a bounded polyhedral domain and $\mathscr{J} = (0, T_{\text{end}})$ be the time interval, where T_{end} is the final time. The evolution of temperature T in $\mathscr{J} \times \bar{\Omega}$ is governed by the problem

$$\frac{\partial T}{\partial t} = \Delta T \qquad \text{in } \mathscr{J} \times \Omega, \tag{18}$$

$$T|_{\partial\Omega} = T_{\text{wall}} \qquad \text{on } \mathscr{J} \times \partial\Omega, \tag{19}$$

$$T|_{t=0} = T_{\text{ini}} \qquad \text{in } \Omega, \tag{20}$$

where (18) is the heat equation (with heat conductivity equal to 1, for simplicity), (19) is a constant Dirichlet boundary condition and (20) is the initial condition. Given a mesh $\mathscr{T}$ covering Ω (recall Definition 1), we integrate (18) over each control volume $K \in \mathscr{T}$, apply the Gauss–Green theorem and use FVM approximations to arrive at

$$\int_K \frac{\partial T}{\partial t}(t,\boldsymbol{x})\mathrm{d}\boldsymbol{x} = \int_{\mathscr{E}_K} \nabla T(t,\boldsymbol{x}) \cdot \boldsymbol{n}\mathrm{d}S \qquad \forall K \in \mathscr{T}, \tag{21}$$

$$\downarrow \qquad\qquad \downarrow$$

$$m(K)\frac{\mathrm{d}T_K}{\mathrm{d}t}(t) = \sum_{\sigma \in \mathscr{E}_K} m(\sigma) F_{K,\sigma} \qquad \forall K \in \mathscr{T}, \tag{22}$$

where:

- T_K is the approximation of $T(t, \boldsymbol{x}_K)$;
- $F_{K,\sigma}$ is the approximation of $\nabla T(t, \boldsymbol{y}_\sigma) \cdot \boldsymbol{n}$ defined as

$$F_{K,\sigma} = \begin{cases} \frac{T_L - T_K}{|\boldsymbol{x}_L - \boldsymbol{x}_K|} & \sigma = \bar{K} \cap \bar{L}, L \in \mathscr{T}, \\ \frac{T_{\text{wall}} - T_K}{|\boldsymbol{y}_\sigma - \boldsymbol{x}_K|} & \sigma \subset \partial\Omega; \end{cases}$$

- $\boldsymbol{x}_K, \boldsymbol{x}_L$ are the centers of the volumes K, L, respectively;
- $\boldsymbol{y}_\sigma$ is the center of σ.

The semidiscrete scheme (22) represents a system of ODEs in the form (17), which is solved by the RKM solver described in Section 5.4.

6.2. Single-Threaded Version

The aim of the single-threaded application is to provide a general overview of the necessary steps related to the generic use of the GTMesh library. The application consists of two main parts: the `RKMSolver()` function and the `HeatConductionProblem` class. The `HeatConductionProblem` has three member functions:

- `loadMesh()`, responsible for loading an unstructured mesh into the `UnstructuredMesh` structure mesh;
- `exportMeshAndData()`, responsible for exporting the computational mesh together with the data;
- `calculateRHS()`, responsible for calculating the right-hand side of (22), which is in the form (17). This function is called from within the `RKMSolver()` function.

6.3. OpenMP Multi-Threaded Version Using Graph Coloring

When using OpenMP for multi-threaded execution, the parallel version of the algorithm can take advantage of edge coloring (see Section 4.3). This approach requires the identification of the operations that access the same memory at the same time. In this particular problem, it is the addition or subtraction of the temperature delta, as shown

on lines 37, 38 in Listing 7. Hence, the coloring has to be calculated for the edges, with respect to the connection to the cells. Then, iterating over the edges with the same color index eliminates the risk of accessing the same cell by multiple threads at the same time, because the edges with the same color do not share any cell. The corresponding logic is incorporated in the `calculateRHS()` function, which is called from within the parallel version of `RKMSolver()`.

Listing 7. Implementation of the heat conduction equation, as introduced in (22). The defined problem object is then passed to `RKMSolver()`, depicted in Listing 6. Note the definition of class traits for the `ComputationData` class in line 4: due to this feature, the data can be serialized/deserialized to/from the VTK format. In addition, `RKMSolver()` can be applied to this problem, as the definition of all necessary arithmetical operations is guaranteed. Lines 55–58 demonstrate that the changes required for transition between 2D and 3D problems are minimal.

```
1  struct ComputationData {
2      double T; //!< temperature
3  };
4  MAKE_NAMED_ATTRIBUTE_TRAIT(ComputationData, "temperature", T);
5
6  // Auxiliary mesh data
7  struct FaceData {
8      double measureOverCellsDistance;
9      double measure;
10 };
11 struct CellData {
12     double invCellVolume;
13 };
14
15 template<unsigned int ProblemDimension>
16 struct HeatConductionProblem{
17     using MeshType = UnstructuredMesh<ProblemDimension, size_t, double>;
18     using ResultType = ComputationData;
19     using ProblemDataContainerType = MeshDataContainer<ResultType, ProblemDimension>;
20     std::shared_ptr<MeshReader<ProblemDimension>> meshReader;
21
22     MeshType mesh;
23     const double T_wall = 300;
24     MeshDataContainer<std::tuple<CellData, FaceData>,ProblemDimension, ProblemDimension-1> meshData;
25
26     void calculateRHS(double time, //time is unused in this problem
27                       const ProblemDataContainerType &compData,
28                       ProblemDataContainerType &outDeltas){
29         for (const auto& cell : mesh.getCells()){
30             outDeltas[cell].T = 0;
31         }
32         for (const auto& face : mesh.getFaces()){
33             const auto cRI = face.getCellRightIndex(), cLI = face.getCellLeftIndex();
34             if (!isBoundaryIndex(cRI) and !isBoundaryIndex(cLI)){
35                 const auto &cR = mesh.getCells().at(cRI), &cL = mesh.getCells()[cLI];
36                 const auto dT_dn = (compData.at(cL).T - compData.at(cR).T) * meshData[face].measureOverCellsDistance;
37                 outDeltas.at(cL).T -= dT_dn;
38                 outDeltas.at(cR).T += dT_dn;
39             } else if (isBoundaryIndex(cLI)) {
40                 const auto &cR = mesh.getCells().at(cRI);
41                 const auto dT_dn = (T_wall - compData[cR].T) * meshData[face].measureOverCellsDistance;
42                 outDeltas.at(cR).T += dT_dn;
43             } else {
44                 const auto &cL = mesh.getCells().at(cLI);
45                 const auto dT_dn = (compData[cL].T - T_wall) * meshData[face].measureOverCellsDistance;
46                 outDeltas.at(cL).T -= dT_dn;
47             }
48         }
49     }
50     // other functions responsible for loading and exporting data
51     // on loading mesh, the auxiliary mesh data are pre-calculated
52 };
53
54 int main() {
55     HeatConductionProblem<3> hcp;              // 3D version
56 //    HeatConductionProblem<2> hcp;             // 2D version
57     auto compData = hcp.loadMesh("../Meshes/mesh3D.vtk");   // 3D version
58 //    auto compData = hcp.loadMesh("../Meshes/mesh2D.vtk"); // 2D version
59     for (int i = 0; i < 10; ++i) {
60         RKMSolver(hcp, compData, 1e-3, i, i + 1.0, 1e-4);
61         // export data ...
62     }
63 }
```

6.4. OpenMP Multi-Threaded Version Using Auxiliary Data

This variant avoids race conditions in cell data access, by introducing auxiliary data structures storing temporary results. As discussed in Section 6.3, the risk of conflict occurs while adding deltas to cell data. Hence, this approach stores the deltas in an auxiliary data structure mapped to the edges. Then, in a separate parallel cycle, the deltas are summed over the cells' boundaries.

7. Conclusions

GTMesh is a C++ library providing data structures and algorithms that facilitate the development of numerical schemes working with general polytopal meshes. In this paper, an overview of the features and design of GTMesh is presented. Aside from the software point of view, the description of general-topology polytopal meshes in arbitrary spatial dimensions is based on the elements of graph theory, which allow for devising the most suitable data structure for mesh representation.

From bottom to top, GTMesh is built upon template metaprogramming principles, using template recursion, specialization, SFINAE and other advanced techniques. Compared to the several alternative projects dealing with similar topics, we believe that GTMesh offers a unique combination of generality, simplicity, efficiency, innovation and elegance that could be beneficial for the developers of numerical software. In particular:

- Despite being a relatively compact project, GTMesh provides the tools for developing complete numerical solvers in a dimension-agnostic manner.
- The code generated by template instantiation for the particular situation is equivalent to a direct implementation with no additional overhead.
- GTMesh not only includes data structures for arbitrary-dimensional meshes but also provides robust algorithms related to mesh geometry.
- The implementation of class reflection in C++ using the Traits mechanism is sufficiently versatile and suitable for development of numerical and data I/O algorithms working with generic data types.
- GTMesh algorithms provide support for multi-threaded (OpenMP) parallelization.

On the other hand, the current implementation of GTMesh has several limitations, in comparison to other much larger projects, such as PUMI, MOAB or TNL:

- Only conforming meshes are supported. Other types of mesh topology, such as recursively refined non-conforming meshes based on quadtree/octree structures, would require a separate implementation.
- GTMesh does not offer support for distributed computing. However, the architecture of GTMesh could be extended in this manner, without the need to redesign its core data structures.
- The authors of GTMesh and this article also collaborate with the developers of TNL [29], opening up the possibility of introducing GPU support, which is currently missing. Some preliminary steps have already been made in this direction.

GTMesh is open-source. Its source code, the code of the sample solver presented in Section 6 and an introductory documentation are publicly available (see Data Availability Statement below). It has also been successfully utilized in the numerical solver for multiphase flow problems [40].

Author Contributions: Conceptualization, T.J. and P.S.; methodology, T.J. and P.S.; software, T.J.; validation, T.J. and P.S.; formal analysis, T.J. and P.S.; investigation, T.J. and P.S.; writing—original draft preparation, T.J. and P.S.; writing—review and editing, T.J. and P.S.; visualization, T.J. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the projects Research Centre for Low-Carbon Energy Technologies (Reg. No. CZ.02.1.01/0.0/0.0/16_019/0000753) and Centre of Advanced Applied Sciences (Reg. No. CZ.02.1.01/0.0/0.0/16_019/0000778), co-financed by the European Union, and by Grant No. SGS23/188/OHK4/3T/14 of the Grant Agency of the Czech Technical University in Prague.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The GTMesh library source code is publicly available under the MIT License at https://github.com/Tomas-Jakubec/GTMesh. The example application presented in Section 6 can be obtained from https://github.com/Tomas-Jakubec/GTMesh-examples. Basic documentation is readily available through the project website at GitHub. In addition, a more detailed explanation of GTMesh design is available in the master's thesis [40] available for download at http://hdl.handle.net/10467/98466.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the following: the design of the study; the collection, analyses or interpretation of data; the writing of the manuscript; the decision to publish the results.

Nomenclature

Important notation used throughout this manuscript:

$\mathbb{A}_G$	adjacency matrix of the graph G (Definition 5)
$\mathbb{A}_{G_{\mathscr{T}^*}}^{d_1,d_2}$	connection matrix of the graph $G_{\mathscr{T}^*}$ from dimension d_1 to dimension d_2
$\mathscr{E}$	a set of faces of all cells in the mesh $\mathscr{T}$ (Definition 1)
$\mathscr{E}_K$	a set of faces constituting the boundary of $K \in \mathscr{T}$ (Definition 1)
$E_{\mathscr{T}^*}$	edges of $G_{\mathscr{T}^*}$, given by (3)
$E_{\mathscr{T}^*}^{d_1,d_2}$	a subset of edges of $G_{\mathscr{T}^*}$ from dimension d_1 to dimension d_2, given by (4)
$G_{\mathscr{T}^*}$	graph representation of $\mathscr{T}^*$
$\mathscr{J}$	time interval (Section 6)
$m(S)$	d-dimensional Lebesgue measure of $S \subset \mathbb{R}^d$ (Section 2)
$\bar{m}(S)$	$(d-1)$-dimensional Hausdorff measure of $S \subset \mathbb{R}^d$ (Section 2)
$N_{\mathscr{T}}^k$	number of elements of dimension $k \in \{0, 1, \ldots, d\}$ in $\mathscr{T}^*$
$N(e)$	neighborhood of the mesh element $e \in \mathscr{T}^*$
$N^k(e)$	elements with dimension $k \in \{0, 1, \ldots, d\}$ connected to $e \in \mathscr{T}^*$
$N_{G_{\mathscr{T}^*}}^{d_1,d_2}(e)$	set of neighbors of dimension d_2 connected by elements of dim. d_1, given by (11)
Ω	spatial domain discretized by the mesh $\mathscr{T}$
$\mathscr{T}$	the mesh, i.e., the set of cells covering Ω (Definition 1)
$\mathscr{T}^*$	the system of geometrical elements of $\mathscr{T}$ (Definition 2)
$\mathscr{T}^k$	system of geometrical elements of $\mathscr{T}$ with dimension k (Definition 2)
T	temperature (in the example problem in Section 6)
$V_{\mathscr{T}^*}$	vertices of $G_{\mathscr{T}^*}$, given by (1)
$\boldsymbol{x}_e$	geometrical center of the element $e \in \mathscr{T}^*$

Acronyms used in this manuscript:

CFD	Computational Fluid Dynamics
DUNE	Distributed and Unified Numerics Environment [28]
FEM	Finite Element Method
FVM	Finite Volume Method
I/O	Input/Output
JSON	JavaScript Object Notation, a lightweight data-interchange format (www.json.org)
MOAB	Mesh-Oriented datABase [26]
ODE	Ordinary Differential Equation
OpenMP	Open Multi-Processing, a parallel programming API (www.openmp.org)
PUMI	Parallel Unstructured Mesh Infrastructure [23]
RKM	Runge–Kutta–Merson, an ODE solver with adaptive time step [38]
SFINAE	Substitution Failure Is Not An Error [31]
TNL	Template Numerical Library [29]
VTK	Visualization Toolkit [36]

References

1. Johnson, C. *Numerical Solution of Partial Differential Equations by the Finite Element Method*; Cambridge University Press: Cambridge, UK, 1987.
2. Szabó, B.; Babuška, I. *Finite Element Analysis—Method, Verification, and Validation*, 2nd ed.; Wiley Series in Computational Mechanics; Wiley: Hoboken, NJ, USA, 2021.
3. Larson, M.G.; Bengzon, F. *The Finite Element Method: Theory, Implementation, and Applications*, 6th ed.; Number 10 in Texts in Computational Science and Engineering; SI Version; Springer: Berlin/Heidelberg, Germany, 2013.
4. Logan, D.L. *A First Course in the Finite Element Method*; CENGAGE: Boston, MA, USA, 2021.

5. Eymard, R.; Gallouët, T.; Herbin, R. Finite Volume Methods. In *Handbook of Numerical Analysis*; Ciarlet, P.G., Lions, J.L., Eds.; Elsevier: Amsterdam, The Netherlands, 2000; Volume 7, pp. 715–1022.
6. Moukalled, F.; Mangani, L.; Darwish, M. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*; Springer: Berlin/Heidelberg, Germany, 2016.
7. Blazek, J. *Computational Fluid Dynamics: Principles and Applications*; Butterworth-Heinemann: Oxford, UK, 2015.
8. Vítek, O.; Macek, J.; Doleček, V.; Syrovátka, Z.; Pavlovic, Z.; Priesching, P.; Tap, F.; Goryntsev, D. Application of advanced combustion models in internal combustion engines based on 3-D CFD LES approach. *Acta Polytech.* **2021**, *61*, 14–32. [CrossRef]
9. Beneš, M.; Strachota, P.; Máca, R.; Havlena, V.; Mach, J. A Quasi-1D Model of Biomass Co-Firing in a Circulating Fluidized Bed Boiler. In *Finite Volumes for Complex Applications VII—Elliptic, Parabolic, and Hyperbolic Problems, Proceedings of the FVCA 7, Berlin, Germany, 15–20 June 2014*; Springer Proceedings in Mathematics & Statistics; Fuhrmann, J., Ohlberger, M., Rohde, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 78, pp. 791–799.
10. Wang, W.; Cao, Y.; Okaze, T. Comparison of hexahedral, tetrahedral and polyhedral cells for reproducing the wind field around an isolated building by LES. *Build. Environ.* **2021**, *195*, 107717. [CrossRef]
11. Sosnowski, M.; Krzywanski, J.; Gnatowska, R. Polyhedral meshing as an innovative approach to computational domain discretization of a cyclone in a fluidized bed CLC unit. *ES3 Web Conf.* **2017**, *14*, 01027. [CrossRef]
12. Sosnowski, M.; Krzywanski, J.; Grabowska, K.; Gnatowska, R. Polyhedral meshing in numerical analysis of conjugate heat transfer. *EPJ Web Conf.* **2018**, *180*, 02096. [CrossRef]
13. Thomas, M.L.; Longest, P.W. Evaluation of the polyhedral mesh style for predicting aerosol deposition in representative models of the conducting airways. *J. Aerosol Sci.* **2022**, *159*, 105851. [CrossRef] [PubMed]
14. Jasak, H.; Jemcov, A.; Tukovic, Z. OpenFOAM: A C++ library for complex physics simulations. In Proceedings of the International Workshop on Coupled Methods in Numerical Dynamicss, Dubrovnik, Croatia, 19–21 September 2007; IUC Dubrovnik: Dubrovnik, Croatia, 2007; Volume 1000, pp. 1–20.
15. Hahn, J.; Mikula, K.; Frolkovič, P.; Basara, B. Finite volume method with the Soner boundary condition for computing the signed distance function on polyhedral meshes. *Int. J. Numer. Methods Eng.* **2022**, *123*, 1057–1077. [CrossRef]
16. Hahn, J.; Mikula, K.; Frolkovič, P.; Basara, B. Inflow-Based Gradient Finite Volume Method for a Propagation in a Normal Direction in a Polyhedron Mesh. *J. Sci. Comput.* **2017**, *72*, 442–465. [CrossRef]
17. Perumal, L. A Brief Review on Polygonal/Polyhedral Finite Element Methods. *Math. Prob. Eng.* **2018**, *2018*, 5792372. [CrossRef]
18. Rashid, M.M.; Selimotic, M. A three-dimensional finite element method with arbitrary polyhedral elements. *Int. J. Numer. Methods Eng.* **2006**, *67*, 226–252. [CrossRef]
19. Bishop, J.E.; Sukumar, N. Polyhedral finite elements for nonlinear solid mechanics using tetrahedral subdivisions and dual-cell aggregation. *Comput. Aided Geom. Des.* **2020**, *77*, 101812. [CrossRef]
20. Nguyen-Ngoc, H.; Cuong-Le, T.; Nguyen, K.D.; Nguyen-Xuan, H.; Abdel-Wahab, M. Three-dimensional polyhedral finite element method for the analysis of multi-directional functionally graded solid shells. *Compos. Struct.* **2023**, *305*, 116538. [CrossRef]
21. Wicke, M.; Botsch, M.; Gross, M. A Finite Element Method on Convex Polyhedra. *Comput. Graph. Forum* **2007**, *26*, 355–364. [CrossRef]
22. Botsch, M.; Steinberg, S.; Bischoff, S.; Kobbelt, L. OpenMesh—A generic and efficient polygon mesh data structure. In Proceedings of the 1st OpenSG Symposium, Darmstadt, Germany, 29 January 2002; IEEE Press: Manhattan, NY, USA, 2002.
23. Ibanez, D.A.; Seol, E.S.; Smith, C.W.; Shephard, M.S. PUMI: Parallel unstructured mesh infrastructure. *ACM Trans. Math. Softw.* **2016**, *42*, 1–28. [CrossRef]
24. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.0. 2021. Available online: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (accessed on 13 July 2023)
25. Rudolf, F.; Rupp, K.; Weinbub, J. *ViennaGrid 2.1.0—User Manual*; Techreport; Vienna University of Technology: Vienna, Austria, 2014.
26. Tautges, T.J.; Ernst, C.; Stimpson, C.; Meyers, R.J.; Merkley, K. *MOAB: A Mesh-Oriented Database*; Technical Report; Sandia National Laboratories: Albuquerque, NM, USA, 2004. [CrossRef]
27. Klinkovský, J.; Oberhuber, T.; Fučík, R.; Žabka, V. Configurable Open-source Data Structure for Distributed Conforming Unstructured Homogeneous Meshes with GPU Support. *ACM Trans. Math. Softw.* **2022**, *48*, 1–30. [CrossRef]
28. Bastian, P.; Blatt, M.; Dedner, A.; Engwer, C.; Klöfkorn, R.; Kornhuber, R.; Ohlberger, M.; Sander, O. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing* **2008**, *82*, 121–138. [CrossRef]
29. Oberhuber, T.; Klinkovský, J.; Fučík, R. TNL: Numerical Library for Modern Parallel Architectures. *Acta Polytech.* **2021**, *61*, 122–134. [CrossRef]
30. *ISO /IEC 14882:2014*; Information Technology—Programming Languages—C++, 4th ed. ISO: London, UK; 2017; p. 1358.
31. Vandevoorde, D.; Josuttis, N.M.; Gregor, D. *C++ Templates: The Complete Guide*, 2nd ed.; Addison-Wesley: Boston, MA, USA, 2017.
32. Meyer, B. *Object-Oriented Software Construction*; Prentice Hall: Upper Saddle River, NJ, USA, 1988.
33. Bondy, J.A.; Murty, U.S.R. *Graph Theory with Applications*; Macmillan London: London, UK, 1976; Volume 290.
34. Brenner, S.; Scott, R. *The Mathematical Theory of Finite Element Methods*; Springer Science & Business Media: Berlin, Germany, 2007; Volume 15.
35. Hahn, J.; Mikula, K.; Frolkovič, P.; Medl'a, M.; Basara, B. Iterative inflow-implicit outflow-explicit finite volume scheme for level-set equations on polyhedron meshes. *Comput. Math. Appl.* **2019**, *77*, 1639–1654.
36. Schroeder, W.; Martin, K.; Lorensen, B. *The Visualization Toolkit*, 4th ed.; Kitware: New York, NY, USA, 2006.
37. *ISO /IEC 14882:2017*; Information Technology—Programming Languages—C++, 5th ed. ISO: London, UK; 2017; p. 1605.

38. Butcher, J.C. *Numerical Methods for Ordinary Differential Equations*, 2nd ed.; Wiley: Chichester, UK, 2008. [CrossRef]
39. Christiansen, J. Numerical solution of ordinary simultaneous differential equations of the 1st order using a method for automatic step change. *Numer. Math.* **1970**, *14*, 317–324. [CrossRef]
40. Jakubec, T. Numerical Simulation of Multiphase Flow on 3D Unstructured Meshes with an Arbitrary Topology. Master's Thesis, Czech Technical University in Prague, Prague, Czech Republic, 2021.