Managing Three-Dimensional Grids and Moving Inner Surfaces

Giovanni Novielli, Silvia Sorbello

766383, 765222
Contents

1 The Public Interface ........................................ 3
  1.1 Reading the TDR file ...................................... 3
  1.2 Writing the public interface ............................. 12
    1.2.1 Class agr_vertex .................................. 12
    1.2.2 Class agr_edge .................................... 13
    1.2.3 Class agr_element .................................. 14
    1.2.4 Class agr_region ................................... 16
    1.2.5 Class agr_bulk ..................................... 17
    1.2.6 Class agr_contact .................................. 18
    1.2.7 Class agr_regioninterface .......................... 19
    1.2.8 Class agr_mesh ..................................... 20
    1.2.9 Class agr_data ..................................... 32

2 The Moving Interface Algorithm ......................... 39
  2.1 The ball-vertex method .................................. 39
    2.1.1 Notation and definition of the problem .............. 40
    2.1.2 The basic edge spring method ....................... 40
    2.1.3 The ball-vertex spring method ...................... 41
    2.1.4 The incremental displacement algorithm .......... 43
  2.2 The moving_interface method ............................ 43
  2.3 Simulation results ....................................... 50
    2.3.1 Numerical validation ................................ 50
    2.3.2 Application examples ............................... 55
Introduction

This project stems from the request by the company where we are doing an internship to create a C++ library that would allow employees to work without the need to use the commercial software and that could be easily modified to suit their requirements.

The aim of our work was to create an interface which allows the user to interact with the mesh and to have access to its information.

The work is divided in two parts. In the first part we created a public interface, consisting of classes, methods and functions which read and process the information stored in a grid file and make them available for users. In the second part we developed an algorithm which realizes the movement of an internal mesh interface.

The outline of this relation is as follows:

- In Chapter 1 we describe the classes which contain the information about the mesh geometry and data, the methods/functions to access and handle it.
- In Chapter 2 we describe the theoretical moving interface algorithm and its practical implementation. Then we show the simulations results.
Chapter 1

The Public Interface

We have been asked to provide a tool for reading and processing input grid files (in TDR format). The access to mesh information must be granted with the same public interface of the commercial software currently in use, in order to avoid rewriting existing code. We carried out this task by creating the C++ library `agr_mesh` which can handle three dimensional meshes of tetrahedral elements.

In this chapter we will explain how we read the input grid file, how we processed and stored the mesh information and how we made it available for users.

1.1 Reading the TDR file

The creation of the 3D memory device’s geometry and the mesh on it is completely handled by the commercial software (Synopsys Sentaurus Mesh) which produces as output a a file with extension `.tdr`.

A Tagged Data Representation (TDR) file is a storage framework to represent the information which need to be visualized. Each TDR file is a collection of 1D, 2D or 3D geometries of different types. Each geometry is composed of regions and contains a set of states. A state contains a set of datasets. Typically states are used to store the simulation state at different instants in time of a transient simulation. A dataset contains the values of a particular quantity for one region. As a consequence, a state typically contains multiple datasets for one quantity, because each dataset contains the values for only one specific region. The layout of the data value inside a dataset depends on the region and geometry types, and on the properties of the dataset. Tags and tag group are used to associate arbitrary additional data to individual objects of a TDR file. This data is structured in a hierarchical way. A tag group corresponds to a directory which contains tags and tag groups. Tags corresponds to files in this analogy. A tag is a “name = value” pair. While the name is always a string, the value can be of different structure and value type. Available structure types are:

- Scalar
- Vector
- Matrix

Available value types are:

- Boolean
- 32-bit Integer
- 64-bit Integer
- Float
Tag groups can be associated with TDR objects of the following types:

- File (also called “Collection”)
- Geometry
- Region
- State
- Dataset

Since a TDR file is essentially a Hierarchical Data Format file (extension `.h5`), we used the HDF5 library in order to read the commercial software output file.

**HDF5 Library**

The Hierarchical Data Format (HDF) implements a model for managing and storing data. This model includes an abstract data model and an abstract storage model (the data format), and libraries to implement the abstract model and to map the storage model to different storage mechanisms. The HDF5 library provides a programming interface to a concrete implementation of the abstract model. The library also implements a model of data transfer, i.e., efficient movement of data from one stored representation to another stored representation. We used the version 1.8.10 of the HDF5 library.

**The Abstract Data Model**  The Abstract Data Model (AMD) is a conceptual model of data, data types and data organization. The AMD defines concepts for defining and describing complex data stored in files. Many kind of data can be mapped to objects of the AMD, and therefore stored and retrieved using HDF5. The key concepts includes:

**Files**

An HDF5 file is a container for an organized collection of objects. These objects are groups, datasets or others. The objects are organized as a rooted, direct graph. Every HDF5 file has at least one object, the root group. All objects are members of the root group or descendents of the root group. HDF5 objects have a unique identity within a single HDF5 file and can be accessed only by its names within the hierarchy of the file.

**Group**

An HDF5 group is analogous to a file system directory and it contains zero or more objects. Every object must be a member of at list one group.

**Dataset**

An HDF5 dataset is a multidimensional array of data elements. The shape of the array (number of dimensions, size of each dimension) is described by the dataspace object. A data element is a single unit of data which may be a number, a character, an array of numbers or characters, or a record of heterogeneous data elements. A data element is a set of bits. The layout of the bits is described by the datatype. The dataspace and datatype are set when the dataset is created, and they cannot be changed for the life or the dataset. The dataset object manages the storage and access to the data. While the data is conceptually a contiguous rectangular array, its actual storage may be a set of compressed chunks, and the access may be through different storage mechanism and caches.
Datasp ac e

An HDF5 dataspace describes the layout of the elements of a multidimensional array. Conceptually, the array is a hyper-rectangle with one to 32 dimensions. HDF5 dataspace can be extendable. Therefore each dimension has a current size and a maximum size, and the maximum may be unlimited. Dataspace are also used to describe hyperslab selections from a dataset. Any subset of the elements of a dataset can be selected for read or write by specifying a set of the hyperslab. A non-rectangular region can be selected by the union of several dataspaces.

Datatype

An HDF5 datatype object describes the layout of a single data element. Data types are categorized into 11 classes of datatype. Each class is interpreted according a set of rules and has a specific set of properties to describe its storage. Thus, the datatype class tells what the element means, and the datatype describes how it is stored. There are two basic types of datatypes, atomic and composite datatypes. Atomic datatypes are indivisible, while composite datatypes are composed of multiple elements of atomic datatypes. In addition to standard types, users can define additional datatypes. Each dataset or attribute has a single datatype object associated with it.

Attribute

Any HDF5 named data object (group, dataset or named datatype) may have zero or more user defined attributes. Attributes are used to document the object. The attributes of an object are stored with the object. An HDF5 attribute has a name and data. The data portion is similar in structure to a dataset: a dataspace defines the layout of an array of data elements, and a datatype defines the storage layout and interpretation of the elements. However an attribute, compared to a dataset, has limitations. An attribute can only be accessed via the object to which it is associated. The data of an attribute must be read or write in a single access, partial reading or writing is not allowed. Attributes do not have attributes. The name, dataspace, and datatype of an attribute are specified when it is created and cannot be changed over the life of the attribute.

PropertyList

HDF5 has a generic property list object. Each list is a collection of name-value pairs. Each class of property list has a specific set of properties. Each property has an implicit name, a datatype, and a value. Property lists are attached to the object in the library and can be used by any part of the library. Some properties are permanent, others are transient. Property lists are information relevant to the behavior of the library, and they are used to control optional behavior for file creation, file access, dataset creation, dataset transfer (read,write), and file mounting.

The Abstract Storage Model

The HDF5 File Format Specification defines how HDF5 objects and data are mapped to a linear address space. The address space is assumed to be a contiguous array of bytes stored on some random access medium. The format defines the standard for how the objects of the abstract data model are mapped to linear addresses. The stored representation is self-describing in the sense that the format defines all the information necessary to read and reconstruct the original objects of the abstract data model. The HDF5 File Format Specification is organized in three parts:

Level 0

The level 0 specification defines the header block for the file. Header block elements include a signature, version information, key parameters of the file layout, and pointers to the rest of the file.

Level 1

The level 1 specification defines the data structures used throughout the file: the B-trees, heaps, and groups.
The level 2 specification defines the data structure for storing the data objects and data.

It is important to realize that the structures defined in the HDF5 file format are not the same as the abstract model. In fact, the object headers, heaps and B-trees of the file specification are not represented in the abstract data model. The format defines a number of objects for managing the storage. The HDF5 File Format Specification defines how the abstract objects (for example groups and datasets) are represented as headers, B-tree blocks, and other elements.

The HDF5 Library The HDF5 library implements the HDF5 abstract data model and storage model. Different tools which can be used on various computational platforms are provided in order to offer a reasonably object-oriented data model and programming interface. To be as portable as possible, the HDF5 Library is implemented in portable C. Although C is not an object-oriented language, the library uses several mechanisms and conventions to implement an object model.

Specifically we used HDFView, a visual tool for browsing and editing HDF5 files and the C++ Application Programming Interfaces.

We will now describe how we managed to read the TDR file:

Step 1: visualization of the TDR file's contents

We had no information about how the mesh data were stored in the TDR file, so we use the tool HDFView to investigate the data organization. Here a sample image of the contents framework:

A typical output file of the commercial software contains a geometry (group geometry_0) in which there are several information:

- **vertex**, a compound dataset consisting of an array \( n_v \times 3 \) (\( n_v \) being the total number of mesh vertices) where the vertices coordinates are stored

- **transformation**, a group containing a matrix A and a vector b which represent the rotation matrix and the shift vector applied to the canonical reference system, usually A is the identity matrix and b is a null vector

- **region_i**, a group associated with the i-th region of the device, it has two important attribute, the name of the region and the material it is made of. It contains the subgroup:
  - **elements_0**, a scalar dataset (one for each region) consisting of an array \( 5n_E \times 1 \) (\( n_E \) being the total number of elements belonging to the region), for each element of the region there are 5 integer numbers, the first one identifies the element type (in our case, tetrahedron), the others are the global indices of the vertices that form the element

- **state_0**, a group containing the datasets with the information about physical quantities defined on the mesh, it has one important attribute, the total number of datasets (which is equal to the number of regions by the number of physical quantities). It contains several subgroups, each of them defined as following:
  - **dataset_i**, a group associated to a single region of the device, containing information about a specific physical quantity, it has five important attributes: the name of the physical quantity and the index of the region to which it refers, the structure type (scalar, vector, tensor), the location type (vertex-centered, element-centered) and the value type (integer, float, double) of the physical quantity to be described. It contains:
values, a scalar dataset consisting of an array of size equal to the product of $I$ to $J$ with

\[
\begin{align*}
I &= n^j_v & \text{if the physical quantity is vertex-centered} \\
I &= n^j_E & \text{if the physical quantity is element-centered} \\
J &= 1 & \text{if the physical quantity is scalar} \\
J &= d & \text{if the physical quantity is a vector} \\
J &= \frac{d(d+1)}{2} & \text{if the physical quantity is tensor}
\end{align*}
\]

where $n^j_v$ is the total number of vertices in the $j-th$ region, $n^j_E$ the total number of elements in the $j-th$ region (assuming the dataset is referred to the $j-th$ region) and $d$ the space dimension (which we assume to be equal to 3), this dataset contains the values which describe the physical quantity on the mesh. At the moment we have considered and implemented only vertex-based physical quantities.

**Step 2: recovery of the TDR file’s contents**

In order to read the TDR file’s contents we wrote a function which performs the information recovery from the TDR file, the function declaration is in the header file `tdr_read.hpp`, while its definition is in the source file, `tdr_read.cpp`.

The signature of the function is as follows

```cpp
int ReadTDR (string TDRname, vector<vector<double> >& coord, 
vector<vector<int> >& bElem, vector<vector<int> >& cElem, 
vector<pair<string,string> >& BulkRegions, 
vector<pair<string,int> >& Contacts, vector<int>& bElemNumber, 
vector<int>& celemNumber)
```

The function returns 0 if everything went well. The only real input is the string representing the name of the TDR file to read, while all other parameters are outputs. As a matter of fact, these vectors are empty and the function will fill them with the mesh information from the TDR file. We will now describe the main steps of the function (for the description of the HDF5 library functions please refer to the HDF5 C++ API Reference Manual).

The first thing we do is open the file in read-only mode.

```cpp
H5File file( TDRname, H5F_ACC_RDONLY );
```

Then we open the outer groups

```cpp
Group collection, geometry;
collection = file.openGroup("collection");
geometry = collection.openGroup("geometry_0");
```

and we read the main attributes: space dimension, number of regions and number of vertices (we show the reading of the space dimension as an example).

```cpp
int dimension, nVertices, nRegions;
Attribute A_dimension, A_nRegions, A_nVertices;
A_dimension = geometry.openAttribute("dimension");
A_dimension.read(PredType::NATIVE_INT32, &dimension);
A_dimension.close();
```

Afterward we read the compound dataset `vertex` column by column (we only show the reading of the coordinates in $x$-direction, the $y$ and $z$ coordinates are read in the same way).

```cpp
double* x(0);
double* y(0);
double* z(0);
```
DataSet D_vertex = geometry.openDataSet("vertex");
x = new double[nVertices];
CompType x_type( sizeof(double) );

Since we are reading only one member (column) of the dataset we have to identify it by its name "x".

x_type.insertMember( "x", 0, PredType::NATIVE_DOUBLE);
D_vertex.read(x, x_type);

Then we fill the vector coord with the mesh vertices coordinates.
coord.resize(nVertices);

We recall that we assume to be in a 3D dimensional space.

for (int i = 0; i < nVertices; i++) {
    coord[i].resize(dimension);
    coord[i][0] = x[i];
    coord[i][1] = y[i];
    coord[i][2] = z[i];
}

Later we take care of the reading of the information about the regions. There are two types of regions in a memory device: bulk regions and contacts. We decided to store the information separately for the bulk regions and the contacts. A Bulk region is a 3D portion of the device, while a contact is a part of the outer surface of a bulk region. Thus in BulkRegions we write the list of the bulk regions, each defined by its name and by the material of which it is made. Though, in Contacts we write the list of the contact regions, each defined by its name and the index of the bulk region to which it is next. In bElemNumber we write, for each bulk region, the number of elements (tetrahedrons) belonging to it. Likewise in cElemNumber we write, for each contact, the number of elements (triangles, since a contact is a 2D surface) belonging to it. In bElem, for each bulk region, we write, for each of its element, the global indices of the 4 vertices that form it. Similarly in cElem, for each contact, we write, for each of its element, the global indices of the 3 vertices that form it.

We will show the recovery of the information from the dataset elements_0 only for the bulk regions, since, for the contacts, the procedure is exactly the same.

int bulk_counter(0), contact_counter(0);
int nBulkRegions(0), nContactRegions(0);
int rType;
vector<int*> bElementVertices;
vector<hsize_t> bElements_VectorLength;
vector<int*> cElementVertices;
vector<hsize_t> cElements_VectorLength;

// Common to all regions
Group G_region;
Attribute A_rName, A_rType;

Since we do not know how many bulk regions there are in the device, we reserve space for the maximum possible number of bulk regions, i.e., the total number of regions.

bElemNumber.reserve(nRegions);
DataSet D_rElements;
Attribute A_rNumElements;
// Only bulk regions;
BulkRegions.reserve(nRegions);
Attribute A_rMaterial;
bElements_VectorLength.reserve(nRegions);
bElementVertices.reserve(nRegions);
Since we do not know how many contacts there are in the device, we reserve space for a maximum number set to 10 contacts.

Contacts.reserve(10);
cElements_VectorLength.reserve(10);
cElementVertices.reserve(10);
Attribute _rBulk0;
cElementNumber.reserve(10);
H5std_string regionIdentifier;

// Loop over the regions
for (int i=0; i < nRegions; i++) {
    regionIdentifier = "region_";
    regionIdentifier += convertInt( i );
    G_region = geometry.openGroup(regionIdentifier);
    // reading of the region type
    A_rType = G_region.openAttribute("type");
    A_rType.read(PredType::NATIVE_INT32,&rType);
    switch(rType) {
    case 0:
        // insertion of the new bulk region
        BulkRegions.push_back(pair<string,string> ());
        // reading of the region name
        A_rName = G_region.openAttribute("name");
        A_rName.read(A_rName.getDataType(),BulkRegions[bulk_counter].first);
        // reading of the region material
        A_rMaterial = G_region.openAttribute("material");
        A_rMaterial.read(A_rMaterial.getDataType(),BulkRegions[bulk_counter].second);
        A_rMaterial.close();
        // reading of number of elements belongin to the region
        bElemNumber.push_back(0);
        D_rElements = G_region.openDataSet("elements_0");
        A_rNumElements = D_rElements.openAttribute("number of elements");
        A_rNumElements.read(PredType::NATIVE_INT32,&bElemNumber[bulk_counter]);
        // reading of the global indices of vertices that form the tetrahedra in the region
        bElements_VectorLength.push_back(0);
        bElementVertices.push_back(0);
        DSP_rElemDataSpace = D_rElements.getSpace();
        DSP_rElemDataSpace.getSimpleExtentDims(&bElements_VectorLength[bulk_counter]);
        bElementVertices[bulk_counter] = new int[bElements_VectorLength[bulk_counter]];
        D_rElements.read(bElementVertices[bulk_counter],PredType::NATIVE_INT32);
        // increase of the bulk regions counter
        bulk_counter++;
        break;
    case 1:
        // contact case
        Contacts.push_back(pair<string,int> ());
        // reading of the contact name
        A_rName = G_region.openAttribute("name");
        A_rName.read(A_rName.getDataType(),Contacts[contact_counter].first);
        // reading of the index of the region to which the contact is next
        A_rBulk0 = G_region.openAttribute("bulk 0");
        A_rBulk0.read(PredType::NATIVE_INT32, &_contacts[contact_counter].second));
// reading of number of elements belongin to the contact and of the global indices of vertices
// that form the triangles in it
...
// increase of the contacts counter
contact_counter++; break;
// case region type = interface (we do not deal with this case)
case 2:
    cerr << "ERROR: please take-out interface with command: struct name !interface: " << rType << endl;
    abort(); break;
// case region type different from all the known valid region types
default:
    cerr << endl << "ERROR: Invalid or not considered region type: " << rType << endl;
    abort();

} // end of the switch case
A_rNumElements.close();
D_rElements.close();
A_rName.close();
A_rType.close();
G_region.close();
} // end of the loop over the regions
nBulkRegions = bulk_counter;

// calculation of the total number of elements in the bulk regions
rElements = accumulate(bElemNumber.begin(),bElemNumber.end(),0);

// writing of bElem
bElem.resize(rElements);
int elem_counter(0);

// loop over the bulk regions
for (int i=0; i<nBulkRegions; i++) {
    unsigned int j=0;
    // loop over the components of the vector bElementVertices
    while (j<bElements_VectorLength[i]) {
        int nVertElem(0);
        if ( (dimension==3) && (bElementVertices[i][j]==5) )
            nVertElem=4;
bElem[elem_counter].resize(nVertElem);
    // loop over the vertices of the tetrahedron
    for (int k=0; k<nVertElem; k++) {
        j++;
bElem[elem_counter][k] = bElementVertices[i][j];
    }
    elem_counter++;
    j++;
} // end of the loop over the components of the vector bElementVertices
delete[] bElementVertices[i];
} // end of the loop over the bulk regions

Finally we process the information obtained to fill the vectors bElem and cElem (we show only the
writing of bElem, since for cElem the procedure is exactly the same).

// calculation of the total number of elements in the bulk regions
rElements = accumulate(bElemNumber.begin(),bElemNumber.end(),0);

// writing of bElem
bElem.resize(rElements);
int elem_counter(0);

// loop over the bulk regions
for (int i=0; i<nBulkRegions; i++) {
    unsigned int j=0;
    // loop over the components of the vector bElementVertices
    while (j<bElements_VectorLength[i]) {
        int nVertElem(0);
        if ( (dimension==3) && (bElementVertices[i][j]==5) )
            nVertElem=4;
bElem[elem_counter].resize(nVertElem);
    // loop over the vertices of the tetrahedron
    for (int k=0; k<nVertElem; k++) {
        j++;
bElem[elem_counter][k] = bElementVertices[i][j];
    }
    elem_counter++;
    j++;
} // end of the loop over the components of the vector bElementVertices
delete[] bElementVertices[i];
} // end of the loop over the bulk regions

11
1.2 Writing the public interface

Once we read and stored the TDR output file data, we organized the mesh information into classes. The declaration of the classes is in the header file `agr_mesh_header.hpp`, while the definition of their members is in different source files. In fact there is a source file for each class. All classes, with the exception of class `agr_data`, are friends of the class `agr_mesh`. We took this decision since it is the constructor of the class `agr_mesh` that processes the data read from the TDR file and saves them into suitable containers. All the mesh information are stored in 6 vectors whose elements are instances of the various classes (vertices, edges, elements, bulk regions, contacts and interfaces). All components of the mesh are interlinked. For instance, a vertex may belong to two different regions, then it is clear that the index of this vertex must appear in the list of vertices of both regions and the indices of the two regions must appear in the vertex list of regions. The vertex index must also appear in the list of vertices of the interface between the two regions. In order to ensure consistency of the links we decide that all connections would be handled by a single function, the constructor of the class `agr_mesh`. Thus the class `agr_mesh` has to be able to modify private members of other classes.

The class `agr_data` contains information about the physical quantities defined on the mesh. We will now describe each class. Many members, both private and public, recur in different classes. Thus, for each class, we will describe only those members whose meaning has not been discussed in previous classes.

1.2.1 Class `agr_vertex`

```cpp
class agr_vertex {
friend class agr_mesh;
public:
    agr_vertex();
    agr_vertex(size_t idx_ext, vector<double> &coords, agr_mesh *mesh_ext);
    agr_vertex(const agr_vertex& orig);
    virtual ~agr_vertex();
    size_t index() const;
    int size_coord() const;
    const double* coord() const;
    size_t size_element() const;
    agr_element* element(size_t i) const;
    size_t size_region() const;
    agr_region* region(size_t i) const;
    size_t size_regioninterface() const;
    agr_regioninterface* regioninterface(size_t i) const;
private:
    size_t idx;
    double* coordinates;
    agr_mesh *mesh;
    vector<int> elem_idx;
    vector<int> reg_idx;
    vector<int> itf_idx;
};
```

The class `agr_vertex` has six private members:

- **idx**: it is an unsigned integer number which represents the global index of the vertex
- **coordinates**: it is an array of three real numbers (we assume to be in a three-dimensional space) which are the coordinates of the vertex
- **mesh**: it is a pointer to the object mesh (instance of the class `agr_mesh`), we use it to connect the vertex to the mesh it belongs to
\textbf{elem_idx} \quad it is a vector of integer numbers, it represents the list of global indices of the elements to which the vertex belongs, (it must contain the index of at least one element)

\textbf{reg_idx} \quad it is a vector of integer numbers, it represents the list of indices of the regions to which the vertex belongs, (it must contain the index of at least one region)

\textbf{itf_idx} \quad it is a vector of integer numbers, it represents the list of indices of the interfaces to which the vertex belongs, (it may be empty)

and thirteen public members:

\texttt{agr\_vertex()} \quad default constructor, it sets \texttt{idx} equal to -1, \texttt{coordinates} and \texttt{mesh} equal to null pointers

\texttt{agr\_vertex(size_t idx\_ext, vector<double> &coords, agr\_mesh *mesh\_ext)} \quad class constructor, it sets \texttt{idx} equal to \texttt{idx\_ext}, \texttt{coordinates} equal to \texttt{coords}, and \texttt{mesh} equal to \texttt{mesh\_ext}

\texttt{agr\_vertex(const agr\_vertex& orig)} \quad copy constructor

\texttt{\~agr\_vertex()} \quad class destructor

\texttt{index()} \quad this method returns the vertex index

\texttt{size\_coord()} \quad this method returns the numbers of coordinates of the vertex (equal to the mesh dimension)

\texttt{coord()} \quad this method returns the pointer \texttt{coordinates} (it does not allow changes to the coordinates values, it gives a read-only access)

\texttt{size\_element()} \quad this method returns the number of elements to which the vertex belongs (equal to the size of \texttt{elem\_idx})

\texttt{element(size_t i)} \quad this method returns the pointer to the \texttt{i}-th element to which the vertex belongs (first it identifies the global index of the \texttt{i}-th element with \texttt{elem\_idx}, then it uses the pointer \texttt{mesh} in order to obtain the corresponding object \texttt{agr\_element})

\texttt{size\_region()} \quad this method returns the number of regions containing the vertex (equal to the size of \texttt{reg\_idx})

\texttt{region(size_t i)} \quad this method returns the pointer to the \texttt{i}-th region to which the vertex belongs (first it identifies the global index of the \texttt{i}-th region with \texttt{reg\_idx}, then it uses the pointer \texttt{mesh} in order to obtain the corresponding object \texttt{agr\_region})

\texttt{size\_regioninterface()} \quad this method returns how many region interfaces the vertex belongs to (equal to the size of \texttt{itf\_idx})

\texttt{regioninterface(size_t i)} \quad this method returns the pointer to the \texttt{i}-th region interface to which the vertex belongs (first it identifies the global index of the \texttt{i}-th region interface with \texttt{itf\_idx}, then it uses the pointer \texttt{mesh} in order to obtain the corresponding object \texttt{agr\_regioninterface})

\subsection{1.2.2 Class \texttt{agr\_edge}}

\begin{verbatim}
class agr_edge {
    friend class agr_mesh;
    public:
    agr_edge();
    agr_edge(size_t idx_ext, endpoints points, agr_mesh *mesh_ext);
    agr_edge(const agr_edge& orig);
    virtual ~agr_edge();

\end{verbatim}
The class `endpoints` is composed of two integers representing the ends of a segment. This class has been used to facilitate the construction of the objects `agr_edge`. The class `agr_edge` has six private members and eleven public members:

- `vert_start` it is an integer number, equal to the global index of the first vertex of the edge
- `vert_end` it is an integer number, equal to the global index of the second vertex of the edge
- `agr_edge()` default constructor, it sets `idx`, `vert_start`, and `vert_end` equal to -1, `mesh` equal to a null pointer
- `agr_edge(size_t idx_ext, endpoints points, agr_mesh *mesh_ext)` class constructor, it sets `idx` equal to `idx_ext`, `vert_start`, and `vert_end` equal to `points`, and `mesh` equal to `mesh_ext`
- `start()` this method returns the pointer to the first vertex of the edge (first it identifies the global index of the first vertex with `vert_start`, then it uses the pointer `mesh` in order to obtain the corresponding object `agr_vertex`)
- `end()` this method returns the pointer to the second vertex of the edge (first it identifies the global index of the first vertex with `vert_end`, then it uses the pointer `mesh` in order to obtain the corresponding object `agr_vertex`)
- `length()` this method returns the length of the edge (e.g., the distance between its two endpoints)

### 1.2.3 Class `agr_element`

```cpp
class agr_element {
    friend class agr_mesh;

public:
    typedef enum { point, line, triangle, rectangle, tetrahedron, pyramid, prism, cuboid, tetrabrick } agr_type;
    agr_element();
    agr_element(size_t idx_ext, agr_type type_ext, int bulk_ext, agr_mesh *mesh_ext);
    agr_element(const agr_element& orig);
    virtual ~agr_element();
    size_t index() const;
    agr_type type() const;
    size_t size_vertex() const;
    agr_vertex* vertex(size_t i) const;
    size_t size_edge() const;
    agr_vertex* start() const;
    agr_vertex* end() const;
    size_t size_element() const;
    agr_element* element(size_t i) const;
    size_t size_region() const;
    agr_region* region(size_t i) const;
    double length() const;
private:
    size_t idx;
    int vert_start;
    int vert_end;
    agr_mesh *mesh;
    vector<int> elem_idx;
    vector<int> reg_idx;
};
```
The class `agr_element` has six private members and twelve public members:

- `mytype` is an integer number representing one of the element types defined as `agr_type`.
- `bulk_idx` is an integer number, it is the global index of the region to which the element belongs to (note that each element can belong to one and only one region).
- `vert_idx` is a vector of integer numbers, it represents the list of global indices of the element vertices (for instance, if the element is a tetrahedron this vector must contain exactly 4 indices).
- `edge_idx` is a vector of integer numbers, it represents the list of global indices of the element’s sides (for instance, if the element is a tetrahedron this vector must contain exactly 6 indices).
- `center` is a vector containing the spatial coordinates of the element circumcenter.
- `agr_element()` default constructor, it sets `idx` and `bulk_idx` equal to -1, `mesh` equal to a null pointer and `mytype` equal to type `point`.
- `agr_element(size_t idx_ext, agr_type type_ext, int bulk_ext, agr_mesh *mesh_ext)` constructor, it sets `idx` equal to idx_ext, `mytype` equal to type_ext, `bulk_idx` equal to bulk_ext and `mesh` equal to mesh_ext.
- `type()` this method returns the type of the element.
- `size_vertex()` this method returns the number of element vertices (equal to the size of `vert_idx`).
- `vertex(size_t i)` this method returns the pointer to the i-th vertex of the element (first it identifies the global index of the i-th vertex with `vert_idx`, then it uses the pointer `mesh` in order to obtain the corresponding object `agr_vertex`).
- `size_edge()` this method returns the number of element edges (equal to the size of `edge_idx`).
- `edge(size_t i)` this method returns the pointer to the i-th edge of the element (first it identifies the global index of the i-th edge with `edge_idx`, then it uses the pointer `mesh` in order to obtain the corresponding object `agr_edge`).
- `measure()` if the element is a tetrahedron this method returns the volume of the element (we do not deal with other types of element since this interface is supposed to be used to perform finite elements simulations).
find_center() this method returns the circumcenter of the element, which is calculated as the point equidistant from the element vertices the first time that this method is called, then the information is stored in center.

radii() this method returns the radius of the element circumscribed sphere (e.g., the distance between the circumcenter and one of the element vertex), and the radius of the element inscribed sphere (calculated with the formula \( r_{in} = \frac{V}{S} \), where \( V \) is the tetrahedron volume and \( S \) its outer surface)

angles() this method returns the minimum and the maximum dihedral angle of the element, (each dihedral angle is computed with the formula \( \gamma = \pi - \arccos(\vec{n}_1 \cdot \vec{n}_2) \), where \( \vec{n}_1 \) and \( \vec{n}_2 \) are the unit normal vectors to the faces forming the angle, directed towards the inside of the element)

1.2.4 Class agr_region

```cpp
class agr_region {
  friend class agr_mesh;
  public:
  agr_region();
  agr_region(size_t idx_ext, string name_ext, agr_mesh *mesh_ext);
  agr_region(const agr_region& orig);
  virtual ~agr_region();
  typedef enum { bulk, contact } agr_type;
  virtual agr_type type() const = 0;
  string name() const;
  size_t size_vertex() const;
  agr_vertex* vertex(size_t i) const;
  protected:
  size_t idx;
  string myname;
  agr_mesh *mesh;
  vector<int> vert_idx;
};
```

The class `agr_region` is a abstract class. In fact the region is an base item whose concrete objects are bulk regions and contacts which will be represented by derived classes of `agr_region`.

The class `agr_region` has 4 protected members and 8 public members:

- `myname` it is a string containing the name of the region
- `vert_idx` it is a vector of integer numbers, it represents the list of global indices of the vertices belonging to the region
- `agr_region()` default constructor, it sets `idx` equal to -1 and `mesh` equal to a null pointer
- `agr_region(size_t idx_ext, string name_ext, agr_mesh *mesh_ext)` class constructor, it sets `idx` equal to `idx_ext`, `myname` equal to `name_ext` and `mesh` equal to `mesh_ext`
- `name()` this method returns the name of the region
- `size_vertex()` this method returns the number of vertices belonging to the region (equal to the size of `vert_idx`)
- `vertex(size_t i)` this method returns the pointer to the i-th vertex of the region (first it identifies the global index of the i-th vertex with `vert_idx`, then it uses the pointer `mesh` in order to obtain the corresponding object `agr_vertex`)
1.2.5 Class agr_bulk

class agr_bulk : public agr_region {
    friend class agr_mesh;
public:
    agr_bulk();
    agr_bulk(size_t idx_ext, string name_ext, string material_ext, agr_mesh *mesh_ext);
    agr_bulk(const agr.bulk& orig);
    virtual ~agr_bulk();
    agr_type type() const;
    string material() const;
    size_t size_element() const;
    agr_element* element(size_t i) const;
    size_t size_regioninterface() const;
    agr_regioninterface* regioninterface(size_t i) const;
    void printBulk(string FILE_NAME) const;
private:
    string mymaterial;
    vector<int> elem_idx;
    vector<int> itf_idx;
};

We recall that a bulk region is a portion of the device, i.e., a 3D block. The class agr.bulk is a
derived class of agr_region. Thus it inherits all the public and protected members of its base class.
Moreover agr.bulk has three private members and eleven public members:

mymaterial it is a string containing the name of the material of which the bulk region is made

elem_idx it is a vector of integer numbers, it represents the list of global indices of the elements
belonging to the bulk region

itf_idx it is a vector of integer numbers, it represents the list of indices of the bulk region’s interfaces

agr.bulk() default constructor, it calls the default constructor of the class agr.region

agr.bulk(size_t idx_ext, string name_ext, string material_ext, agr_mesh *mesh_ext) class
constructor, it sets idx equal to idx_ext, myname equal to name_ext, mymaterial equal to material_ext and mesh equal to mesh_ext

material() this method returns the material of which the bulk region is made

size_element() this method returns the number of elements belonging to the bulk region (equal to
the size of elem_idx )

element(size_t i) this method returns the pointer to the i-th element of the bulk region (first it
identifies the global index of the i-th vertex with elem_idx, then it uses the pointer mesh in
order to obtain the corresponding object agr_element)

size_regioninterface() this method returns the number of interfaces shared by the bulk region
(equal to the size of itf_idx ) with other regions

regioninterface(size_t i) this method returns the pointer to the i-th interface of the bulk region
(first it identifies the global index of the i-th interface with itf_idx, then it uses the pointer
mesh in order to obtain the corresponding object agr_regioninterface)

printBulk(string FILE_NAME) this method produces a VTK ASCII file whose name is FILE_NAME,
the file contains information about the geometry and topology of the bulk region (vertices,
elements), the output file can be loaded with a graphic visualization software (for example
PARAVIEW) to visualize the bulk region
1.2.6 Class agr_contact

```cpp
class agr_contact : public agr_region {
friend class agr_mesh;
public:
agr_contact();
agr_contact(size_t idx_ext, string name_ext, agr_mesh *mesh_ext);
agr_contact(const agr_contact& orig);
virtual ~agr_contact();
agr_type type () const;
size_t size_element() const;
vector<int> const & element(size_t i) const;
size_t size_vertex_patch(size_t vertex_index) const;
vector<int> const & patch_element(size_t vertex_index, size_t triangle_index) const;
double patch_surface(size_t vertex_index) const;
double total_surface() const;
private:
vector<vector<int>> surface_elements;
map<int, vector<int>> vertices_patch;
};
```

We recall that a contact is a part of the outer surface of a bulk region, i.e., a 2D surface. The class agr_contact is a derived class of agr_region. Thus it inherits all the public and protected members of its base class. Moreover agr_contact has one private member and seven public members:

- **surface_elements** it is a matrix $n_c^t \times 3$, $n_c^t$ being the number of triangles belonging to the contact, it lists, for each triangle, the global indices of its 3 vertices.
- **vertices_patch** it is a map which lists for each contact vertex, its global index being the key value of the map, the local indices of the triangles to which the vertex belong.
- **agr_contact()** default constructor, it calls the default constructor of the class agr_region.
- **agr_contact(size_t idx_ext, string name_ext, agr_mesh *mesh_ext)** class constructor, it sets idx_equal to idx_ext, myname equal to name_ext and mesh_equal to mesh_ext.
- **size_element()** this method returns the number of elements belonging to the contact (equal to the size of surface_elements).
- **element(size_t triangle_index)** this method returns a vector containing the global indices of the 3 vertices of the triangle_index-th triangle of the contact.
- **size_vertex_patch(size_t vertex_index)** this method returns the number of triangles belonging to the patch of the vertex whose local index is vertex_index.
- **patch_element(size_t vertex_index, size_t i)** this method returns a vector containing the global indices of the 3 vertices of the i-th triangle of the patch of the vertex whose local index is vertex_index.
- **patch_surface(size_t vertex_index)** this method returns the size of the patch’s area of the vertex whose local index is vertex_index.
- **total_surface()** this method returns the size of the contact total area.
1.2.7 Class agr_regioninterface

class agr_regioninterface {
friend class agr_mesh;
public:
agr_regioninterface();
agr_regioninterface(size_t idx_ext, int bulk_one_ext, int bulk_two_ext, agr_mesh *mesh_ext);
agr_regioninterface(const agr_regioninterface & orig);
virtual ~agr_regioninterface();
size_t index() const;
agr_bulk* bulk1() const;
agr_bulk* bulk2() const;
size_t size_vertex() const;
agr_vertex* vertex(size_t i) const;
size_t size_element() const;
vector<int> const & element(size_t i) const;
size_t size_element() const;
vector< int > const & element(size_t i) const;
size_t size_vertex_patch(size_t vertex_index) const;
vector< int > const & patch_element(size_t vertex_index, size_t triangle_index) const;
double patch_surface(size_t vertex_index) const;
double box_area(size_t triangle, size_t vertex_index) const;
double voronoi_surface(size_t vertex_index) const;
double total_surface() const;
void printRegionInterface(string FILE_NAME) const;
private:
size_t idx;
agr_mesh *mesh;
int bulk_one;
int bulk_two;
vector< int > vert_idx;
vector< vector< int > > surface_elements;
map< int, vector< int > > vertices_patch;
};

We define an interface as a surface shared by two different bulk regions. The class agr_regioninterface has seven private members and eighteen public members:

bulk_one it is an integer number representing the index of the first bulk region to which the interface belongs

bulk_two it is an integer number representing the index of the second bulk region to which the interface belongs

vert_idx it is a vector of integer numbers, it represents the list of global indices of the vertices belonging to the interface

surface_elements it is a matrix \( n_T \times 3 \), \( n_T \) being the number of triangles belonging to the interface, it lists, for each triangle, the global indices of its 3 vertices

agr_regioninterface() default constructor, it sets idx equal to -1 and mesh equal to a null pointer

agr_regioninterface(size_t idx_ext, int bulk_one_ext, int bulk_two_ext, agr_mesh *mesh_ext) class constructor, it sets idx equal to idx_ext, bulk_one equal to bulk_one_ext, bulk_two equal to bulk_two_ext and mesh equal to mesh_ext

bulk1() this method returns the pointer to the first bulk region to which the interface belongs (first it identifies the index of the first bulk region with bulk_one, then it uses the pointer mesh in order to obtain the corresponding object agr_bulk)
bulk2() this method returns the pointer to the second bulk region to which the interface belongs (first it identifies the index of the second bulk region with bulk_two, then it uses the pointer mesh in order to obtain the corresponding object agr_bulk)

box_area(size_t triangle_index, size_t vertex_index) this method returns the area of the box\textsuperscript{1} associated with the vertex whose local index is vertex_index in the triangle_index-th triangle of its patch

voronoi_surface(size_t vertex_index) this method returns the area of the Voronoi surface\textsuperscript{1} associated to the vertex whose local index is vertex_index

printRegionInterface(string FILE_NAME) this method produce a VTK file whose name is FILE_NAME, the file contains information about the geometry and topology of the interface (vertices, elements), the output file can be loaded with a graphic visualization software (for example PARAVIEW) to visualize the interface

1.2.8 Class agr_mesh

```cpp
class agr_mesh {
public:
    agr_mesh();
    agr_mesh(string FILE_NAME);
    agr_mesh(const agr_mesh& orig);
    virtual ~agr_mesh();
    string filename() const;
    int dim() const;
    size_t size_vertex() const;
    agr_vertex* vertex(size_t i) const;
    size_t size_edge() const;
    agr_edge* edge(size_t i) const;
    size_t size_element() const;
    agr_element* element(size_t i) const;
    size_t size_region() const;
    agr_region* region(size_t i) const;
    size_t size_bulk() const;
    agr_bulk* bulk(size_t i) const;
    size_t size_contact() const;
    agr_contact* contact(size_t i) const;
}
```

\textsuperscript{1}Definition of Voronoi surface and box associated to a vertex

We introduce the subgrid fig.(a) on the general triangle $K$ belonging to the interface. We denote by $e_i$ the edge opposite to vertex $v_i$, $i = 1, 2, 3$. Moreover we let $s_i$ be the signed length of the segment joining the midpoint $m_i$ of the edge $e_i$ with the intersection $C_K$ of the perpendicular bisectors to each edge $e_i$ (that is the circumcenter of triangle $K$). We also indicate by $B_i$ the portion of $K$ delimited by $v_i$, $C_K$, the cross-section $s_j$, $s_k$ and the halves of edges $e_j$, $e_k$ and we refer to the control volume $B_i$ as the box associated with $v_i$.

We also define $B_v$, the Voronoi cell associated with every node $v \in T_h$, as the union of the boxes $B_i$ of each triangle that has $v$ as local vertex $v_i$ (with $i = 1, 2, 3$), fig.(b).
size_t size_regioninterface() const;
agr_regioninterface* regioninterface(size_t i) const;
int saveMesh_hdf5(string HDF_FILE_NAME) const;
void printMesh_xmf(string XML_FILE_NAME) const;

private:
int dimension;
string file_name;
vector<agr_vertex> *Vertices;
vector<agr_edge> *Edges;
vector<agr_element> *Elements;
vector<agr_bulk> *Bulks;
vector<agr_contact> *Contacts;
vector<agr_regioninterface> *Interfaces;
};

The class agr_mesh is the class that contains all general information about the mesh. It also allows
the connection among the objects which constitute the grid. The class agr_mesh has eight private
members and twenty-two public members:
dimension it is an integer number representing the space dimension (we assume to be in a 3D space)
FILE_NAME it is a string containing the name of the input file read to get mesh information
Vertices it is a vector of objects belonging to the class agr_vertex, it represents the list of the mesh
vertices
Edges it is a vector of objects belonging to the class agr_edge, it represents the list of the mesh
edges
Elements it is a vector of objects belonging to the class agr_element, it represents the list of the
mesh elements
Bulks it is a vector of objects belonging to the class agr_bulk, it represents the list of the mesh
bulk regions
Contacts it is a vector of objects belonging to the class agr_contact, it represents the list of the
mesh contacts
Interfaces it is a vector of objects belonging to the class agr_regioninterface, it represents the
list of the mesh interfaces
agr_mesh default constructor, it sets dimension equal to -1 and all vectors components equal to 0
agr_mesh(string name_ext) class constructor, it will be discussed later
filename() this method returns the name of the TDR file from which the mesh has been read
dim() this method returns the space dimension
size_vertex() this method returns the total number of vertices belonging to the mesh
vertex(size_t i) this method returns the i-th vertex belonging to the mesh
size_edge() this method returns the total number of edges belonging to the mesh
edge(size_t i) this method returns the i-th edge belonging to the mesh
size_element() this method returns the total number of elements belonging to the mesh
element(size_t i) this method returns the i-th element belonging to the mesh
size_region() this method returns the total number of regions (both bulk regions and contacts) belonging to the mesh

region(size_t i) this method returns the i-th region belonging to the mesh

size_bulk() this method returns the total number of bulk regions belonging to the mesh

bulk(size_t i) this method returns the i-th bulk region belonging to the mesh

size_contact() this method returns the total number of contacts belonging to the mesh

contact(size_t i) this method returns the i-th contact belonging to the mesh

size_regioninterface() this method returns the total number of interfaces belonging to the mesh

regioninterface(size_t i) this method returns the i-th interface belonging to the mesh

saveMesh_hdf5(string HDF_FILE_NAME) this method produces an HDF5 file whose name is HDF_FILE_NAME; this file contains information about the geometry and topology of the mesh (vertices, elements), and can be used (with the output file of the following method) with a graphic visualization software (for example PARAVIEW) to visualize the mesh; the method will be discussed in more detail later

printMesh_xmf(string XML_FILE_NAME) this method produces a XML file whose name is XML_FILE_NAME; this file contains information about the mesh, and can be used (with the output file of the previous method) with a graphic visualization software to visualize the mesh; the method will be discussed in more detail later

**Agr_mesh class constructor** The *agr_mesh* class constructor is the function which processes the data retrieved from the input file and stores them in an organized framework that can be easily exploited by users. We will now describe the main steps of the constructor. The input file can be of type TDR or XDMF (see the paragraph Mesh visualization below).

First the constructor calls the function ReadTDR or ReadXMF. We recall that this function produces seven output vectors:

- **coord** coordinates of the vertices belonging to the mesh
- **BulkRegions** list of the bulk regions, each defined by its name and by the material of which it is made
- **Contacts** list of the contact regions, each defined by its name (the index of the bulk region to which it is next is also stored as information)
- **bElemNumber** list of the number of elements (tetrahedrons) belonging to each bulk region
- **celemNumber** list of the number of elements (triangles) belonging to each contact
- **bElem** list of the global indices of the 4 vertices that form each element of each bulk region
- **cElem** list of the global indices of the 3 vertices that form each element of each contact

Once the input file is read, the vector **Vertices** is filled. For each mesh vertex we create an object *agr_vertex* and we add it to the vector:

```c++
Vertices = new vector<agr_vertex>;
Vertices->reserve(coord.size());
for (unsigned int i=0; i<coord.size(); i++) {
    (*Vertices).push_back(agr_vertex(size_t (i), coord[i], this));
}
```

22
Then we deal with the bulk regions. For each bulk region we create an object `agr_bulk` and we add it to the vector `Bulks`. We read the vector `bElem` and for each bulk region we extrapolate the elements which constitute it. We extrapolate also the edges and vertices belonging to that bulk region.

```cpp
Bulks = new vector<agr_bulk>;
Bulks->reserve(bElemNumber.size());
vector<set<int> > Reg_per_Vert;
pair<set<int>::iterator, bool> temp;
map<endpoints, int, bool (*)(const endpoints&, const endpoints&)> edgesMap(f_pt);
pair<map<endpoints, int>::iterator, bool> temp2;
int start(0), end(0), edges_count(0);
// Loop over the bulk regions
for (unsigned int i=0; i<bElemNumber.size(); i++) {
    // Creation of the object bulk region
    (*Bulks).push_back(agr_bulk(size_t (i), BulkRegions[i].first, BulkRegions[i].second, this));
    // Reserve space for the exact number of elements belonging to the bulk region. Unfortunately
    // we do not know how many vertices belong to the bulk region. Thus we decided to set the maximum
    // number of vertices equal to three times the number of elements of the region.
    (*Bulks)[i].elem_idx.reserve(bElemNumber[i]);
    (*Bulks)[i].vert_idx.reserve(bElemNumber[i]*3);
    start = end;
    end = start + bElemNumber[i];
    // Loop over the elements of the i-th bulk region
    for (int j=start; j<end; j++) {
        agr_element::agr_type type_ext;
        // In order to get the element’s type we use the information about its number of vertices (we recall
        // that we suppose to be in a 3D space).
        switch (bElem[j].size()) {
            case 4: type_ext = agr_element::tetrahedron; break;
            case 5: type_ext = agr_element::pyramid; break;
            case 6: type_ext = agr_element::prism; break;
            case 8: type_ext = agr_element::cuboid; break;
            case 7: type_ext = agr_element::tetrabrick; break;
            default: cerr << " Something wrong with the reading of elements type: the element of index
            " << j << " is not compatible with the dimension!" <<endl; abort(); break;
        }
        // Creation of the object element
        (*Elements).push_back(agr_element(size_t (j), type_ext, int (i), this));
        // Addition of the new element in the bulk region’s list of elements
        (*Bulks)[i].elem_idx.push_back(j);
        // For each element of the bulk region we reserve space for the vertices and the edges, whose number
        // is determined by the element’s type.
        (*Elements)[j].vert_idx.reserve(bElem[j].size());
        (*Elements)[j].edge_idx.reserve((*Elements)[j].size_edge());
        // Loop over the element’s vertices
        for (unsigned int k=0; k<bElem[j].size(); k++) {
            // Addition of the vertex in the element’s list of vertices
            (*Elements)[j].vert_idx.push_back(bElem[j][k]);
            // Addition of the vertex in the element’s list of elements
            (*Vertices)[bElem[j][k]].elem_idx.push_back(j);
        }
    }
}
```
We want to establish a connection between the vertex and the region it belongs to. We cannot simply insert the region index in the vector `reg_idx` of the vertex because in our loops we consider multiple times the same vertex (while, for instance, each element is regarded one single time). To avoid inserting each bulk region index more than once, we use a temporary set to check if the current bulk region has already been added to the vertex list of regions.

```cpp
temp = Reg_per_Vert[bElem[j][k]].insert(int (i));
if (temp.second)
  // Addition of the bulk region in the vertex list of region (ONLY IF it has not been inserted yet)
  (*Bulks)[i].vert_idx.push_back(bElem[j][k]);
} // End of the loop over the element’s vertices
```

We rebuilt the element’s edges. Each edge is created considering a pair of element’s vertices.

```cpp
// Loop over the pairs of element’s vertices
for (unsigned int k=0; k<(*Elements)[j].size_vertex()-1; k++) {
  for (unsigned int k2=k+1; k2<(*Elements)[j].size_vertex(); k2++) {
    // Creation of the object endpoints containing the vertices of the edge
    endpoints points(int((*Elements)[j].vertex(k)->index()), int((*Elements)[j].vertex(k2)->index()));
  }
} // End of the loop over the pairs of element’s vertices
```

We cannot simply insert the object edge in the vector `Edges` because in our loops we consider multiple times the same edge. To avoid inserting each edge more than once we use a temporary map to check if the edge we are considering has already been added to the global list of edges.

```cpp
temp2 = edgesMap.insert(pair< endpoints, int>(points, edges_count));
if (temp2.second) {
  // Creation of the object edge and its addition in the mesh list of edges (ONLY IF it has not been inserted yet)
  (*Edges).push_back(agr_edge(size_t (edges_count), points, this));
}
```

We need to reserve space for the edge list of elements but we do not know how many elements the edge belongs to. Thus we decided to set the maximum number of elements equal to 20.

```cpp
(*Edges)[edges_count].elem_idx.reserve(20);
Reg_per_Edge.push_back(set<int> ());
// Increment of the edges counter (ONLY IF an edge has been inserted)
edges_count++;
```

We want to establish a connection between the edge and the region it belongs to. We cannot simply insert the region index in the vector `reg_idx` of the edge since we may consider multiple times the same edge. To avoid inserting each bulk region index more than once we use a temporary set to check if the current bulk region has already been added to the edge list of regions.

```cpp
temp = Reg_per_Edge[temp2.first->second].insert(int (i));
}
```

Now we fill the vector `reg_idx` for each vertex and each edge using the sets we created on purpose `Reg_per_Vert` and `Reg_per_Edge`, respectively.
for (unsigned int i=0; i<Reg_per_Vert.size(); i++) {
    (*Vertices)[i].reg_idx.resize(Reg_per_Vert[i].size());
    copy(Reg_per_Vert[i].begin(),Reg_per_Vert[i].end(),(*Vertices)[i].reg_idx.begin());
}
for (unsigned int i=0; i<Reg_per_Edge.size(); i++) {
    (*Edges)[i].reg_idx.resize(Reg_per_Edge[i].size());
    copy(Reg_per_Edge[i].begin(),Reg_per_Edge[i].end(),(*Edges)[i].reg_idx.begin());
}

After we dealt with the bulk regions, we want to process the information we have to define the mesh interfaces. We define Interface a pair of two integer numbers representing the indices of the bulk regions which share the interface. We create a vector of boolean values to divide the vertices that have already been regarded from those which still have to be considered. We also create a set to contain temporarily the interfaces that we define. This is necessary because, since we define the interfaces using the vertices, we will recover the same interface multiple times.

    set<Interface> itf_tmp;
    vector<bool> alreadyDoneVertex(Vertices->size(),false);

We consider all the vertices. For each vertex we verify how many bulk regions it belongs to. We recall that, for now, in the vector reg_idx of all vertices there are only bulk regions (since we have not processed contacts yet). If the vertex belongs to one single bulk region it is an internal vertex, thus it is not involved in the interfaces definition. If the vertex belongs to exactly two bulk regions then it lies on the inner surface of an interface.

    // Loop over the mesh vertices
    for (unsigned int i=0; i < (*Vertices).size(); i++) {
        if ( (*Vertices)[i].size_region() == 2 ) {
            // Case 1: the vertex belongs to exactly 2 bulk regions: creation of the interface and its addition to the set itf_tmp
            itf_tmp.insert(Interface((*Vertices)[i].reg_idx[0],(*Vertices)[i].reg_idx[1]) );
        }
    }

    // Loop over the elements to which the i-th vertex belongs
    for (unsigned int j=0; j < (*Vertices)[i].size_element(); j++) {
        agr_element * tmp_element = (*Vertices)[i].element(j);
        // Loop over the pairs of element’s vertices
        for (unsigned int j1=0; j1 < tmp_element->size_vertex()-1; j1++) {
            int tmp_index1 = tmp_element->vertex(j1)->index();
            for (unsigned int j2=j1+1; j2 < tmp_element->size_vertex(); j2++) {
                int tmp_index2 = tmp_element->vertex(j2)->index();

                // We proceed only if the two vertices have not been done yet and if they both belong to 3 or more region, that is, if we have not yet considered the tetrahedron face formed by the vertices whose indices are tmp_index1, tmp_index2 and i, and if this triangle has all vertices potentially belonging to an interface.
            }
        }
    }

    // Case 2: the vertex belongs to 3 or more bulk regions: mark the vertex as already done
    alreadyDoneVertex[i] = true;
    // Loop over the elements to which the i-th vertex belongs
    for (unsigned int j=0; j < (*Vertices)[i].size_element(); j++) {
        agr_element * tmp_element = (*Vertices)[i].element(j);
        // Loop over the pairs of element’s vertices
        for (unsigned int j1=0; j1 < tmp_element->size_vertex()-1; j1++) {
            int tmp_index1 = tmp_element->vertex(j1)->index();
            for (unsigned int j2=j1+1; j2 < tmp_element->size_vertex(); j2++) {
                int tmp_index2 = tmp_element->vertex(j2)->index();

                // We proceed only if the two vertices have not been done yet and if they both belong to 3 or more region, that is, if we have not yet considered the tetrahedron face formed by the vertices whose indices are tmp_index1, tmp_index2 and i, and if this triangle has all vertices potentially belonging to an interface.
if (!alreadyDoneVertex[tmp_index1] && !alreadyDoneVertex[tmp_index2] && (*Vertices)[tmp_index1].size_region() >= 3 && (*Vertices)[tmp_index2].size_region() >= 3) {
    // Loop over the combinations of bulk regions to which the i-th vertex belongs
    for (unsigned int i1=0; i1 < (*Vertices)[i].size_region()-1; i1++) {
        for (unsigned int i2=i1+1; i2 < (*Vertices)[i].size_region(); i2++) {
            // Creation of the interface formed by the pair of bulk regions i1-i2 of the i-th vertex
            Interface temp_itf0((*Vertices)[i].reg_idx[i1], (*Vertices)[i].reg_idx[i2]);
            // Loop over the combinations of bulk regions to which the vertex whose index is tmp_index1 belongs
            for (unsigned int j11=0; j11 < (*Vertices)[tmp_index1].size_region()-1; j11++) {
                for (unsigned int j12=j11+1; j12 < (*Vertices)[tmp_index1].size_region(); j12++) {
                    // Creation of the interface formed by the pair of bulk regions i1-i2 of the vertex whose index is tmp_index1
                    Interface temp_itf1((*Vertices)[tmp_index1].reg_idx[j11], (*Vertices)[tmp_index1].reg_idx[j12]);
                    // Loop over the combinations of bulk regions to which the vertex whose index is tmp_index2 belongs
                    for (unsigned int j21=0; j21 < (*Vertices)[tmp_index2].size_region()-1; j21++) {
                        for (unsigned int j22=j21+1; j22 < (*Vertices)[tmp_index2].size_region(); j22++) {
                            // Creation of the interface formed by the pair of bulk regions i1-i2 of the vertex whose index is tmp_index2
                            Interface temp_itf2((*Vertices)[tmp_index2].reg_idx[j21], (*Vertices)[tmp_index2].reg_idx[j22]);
                            if (temp_itf0 == temp_itf1 && temp_itf1 == temp_itf2)
                                // Addition of the interface to the set itf_tmp ONLY IF all the three vertices belong to the same interface
                                itf_tmp.insert(temp_itf0);
                        }
                    }
                }
            }
        }
    }
}
// End of the loop over the pairs of element's vertices
// End of loop over the elements to which the i-th vertex belongs
// End of the case 2 (vertex belonging to 3 or more bulk regions)

After we wrote the list of interfaces in the set itf_tmp, we scroll through this list to fill the vector Interfaces. For each interface we have to create the object agr_regioninterface. We also connect the interface to the bulk regions which share it.

Interfaces = new vector<agr_regioninterface>;
Interfaces->reserve(itf_tmp.size());
int itf_counter = 0;
// Loop over the mesh interfaces
for (set<Interface>::iterator i=itf_tmp.begin(); i != itf_tmp.end(); i++, itf_counter++) {
    // Creation of the object interface and its addition to the vector interfaces
    (*Interfaces).push_back(agr_regioninterface(itf_counter, i->first, i->second, this));
    // Addition of the interface in the bulk regions list of interfaces
    (*Bulks)[i->first].itf_idx.push_back(itf_counter);
    (*Bulks)[i->second].itf_idx.push_back(itf_counter);
} // End of the loop over the mesh interfaces
Then we have to recover the list of vertices and elements (triangles) belonging to the interface. We create two sets of integers \texttt{vert\_per\_itf} and \texttt{itf\_per\_vert} to handle the connections between vertices and interfaces, since we may consider the same vertex or the same interface multiple times. We also create a vector of boolean values to mark the interfaces whose vertices and elements have already been completely retrieved.

```cpp
set<Interface>::iterator set_ITF_iter;
vector< set<int> > vert_per_itf(itf_tmp.size());
vector< set<int> > itf_per_vert(Vertices->size());
vector<bool> alreadyDoneItf(itf_tmp.size(),false);
```

We loop over the mesh interfaces:

```cpp
for (unsigned int i = 0; i < Interfaces->size(); i++) {
    if (!alreadyDoneItf[i]) {
        // Loop over the elements of the region whose index is \texttt{tmp\_index}
        set<int> itfDoneInThisCycle;
        int tmp_index = (*Interfaces)[i].bulk_one;
        for (unsigned int j=0; j < (*Bulks)[tmp_index].size_element(); j++) {
            agr_element * tmp_element = (*Bulks)[tmp_index].element(j);
            int possibleInterfaceVertices = 0;
            vector<int> candidates(tmp_element->size_vertex());
            for (unsigned int k=0; k < tmp_element->size_vertex(); k++){
                if (tmp_element->vertex(k)->size_region() >= 2) {
                    candidates[possibleInterfaceVertices] = k;
                    possibleInterfaceVertices++;
                }
            }
            if ( possibleInterfaceVertices >= 3) {
                int combinations = 1 + 3 * ( possibleInterfaceVertices == 4);
                for (int comb_count = 0; comb_count < combinations; comb_count++) {
                    agr_vertex * tmp_vertex1; agr_vertex * tmp_vertex2;
                }
            }
        }
    }
}
```
agr_vertex * tmp_vertex3;
switch (comb_count) {
  case 0:
    tmp_vertex1 = tmp_element->vertex(candidates[0]);
    tmp_vertex2 = tmp_element->vertex(candidates[1]);
    tmp_vertex3 = tmp_element->vertex(candidates[2]);
    break;
  case 1:
    tmp_vertex1 = tmp_element->vertex(candidates[0]);
    tmp_vertex2 = tmp_element->vertex(candidates[1]);
    tmp_vertex3 = tmp_element->vertex(candidates[3]);
    break;
  case 2:
    tmp_vertex1 = tmp_element->vertex(candidates[0]);
    tmp_vertex2 = tmp_element->vertex(candidates[2]);
    tmp_vertex3 = tmp_element->vertex(candidates[3]);
    break;
  case 3:
    tmp_vertex1 = tmp_element->vertex(candidates[1]);
    tmp_vertex2 = tmp_element->vertex(candidates[2]);
    tmp_vertex3 = tmp_element->vertex(candidates[3]);
    break;
}

Once we know the three vertices that form the face, we identify which one is the element, besides tmp_element, to which the face belongs.

// Loop over the elements of the first vertex forming the face
for (unsigned int k1=0; k1 < tmp_vertex1->size_element(); k1++) {
    unsigned int tmp_elem_idx(tmp_vertex1->element(k1)->index());
    if (tmp_elem_idx != tmp_element->index() ) {
        if ( tmp_vertex2->elem_idx.find(tmp_elem_idx) != tmp_vertex2->elem_idx.end() ) {
            if ( tmp_vertex3->elem_idx.find(tmp_elem_idx) != tmp_vertex3->elem_idx.end() ) {
                After we found the two elements sharing the face, we go back and we identify the bulk regions to which the elements belong. Then we identify the interface shared by those two regions.

    int firstRegion = tmp_element->bulk_idx;
    int secondRegion = (*Elements)[tmp_elem_idx].bulk_idx;
    set_ITF_iter = itf_tmp.find( Interface(firstRegion,secondRegion) );
    // Recovery of the index of the interface
    int tmp = distance(itf_tmp.begin(),set_ITF_iter);

    If the interface has not been processed yet we have to establish a connection between the vertices of the face and the interface they belong to. We cannot simply insert the interface index in the vector itf_idx of each vertex (or the vertices indices in the vector vert_idx of the interface) since we consider multiple times the same interface (and the same vertices). Thus we use the vectors of set itf_per_vert and vert_per_itf to check if the current interface has already been added to the vertices list of interfaces or if the vertices have already been added to the interface list of vertices.

    We also add the global indices of the vertices which form the face to the interface list of triangles surface_elements. Lastly we add the local index of the triangle to the list of elements belonging to the vertices patches vertices_patch.

    if ( set_ITF_iter != itf_tmp.end() && !alreadyDoneItf[tmp] ) {
        // Addition of the vertices in interface list of vertices
vert_per_itf[tmp].insert( tmp_vertex1->index() );
vert_per_itf[tmp].insert( tmp_vertex2->index() );
vert_per_itf[tmp].insert( tmp_vertex3->index() );
// Addition of the interface in the vertices list of interfaces
itf_per_vert[tmp_vertex1->index()].insert(tmp);
itf_per_vert[tmp_vertex2->index()].insert(tmp);
itf_per_vert[tmp_vertex3->index()].insert(tmp);
// Addition of the triangle to the interface list of elements
(*Interfaces)[tmp].surface_elements.push_back(vector<int>(3);
(*Interfaces)[tmp].surface_elements.back()[0] = tmp_vertex1->index();
(*Interfaces)[tmp].surface_elements.back()[1] = tmp_vertex2->index();
(*Interfaces)[tmp].surface_elements.back()[2] = tmp_vertex3->index();
// Addition of the triangle in the vertices patch of elements
int triangle_position = (*Interfaces)[tmp].surface_elements.size()-1;
(*Interfaces)[tmp].vertices_patch[tmp_vertex1->index()].push_back(triangle_position);
(*Interfaces)[tmp].vertices_patch[tmp_vertex2->index()].push_back(triangle_position);
(*Interfaces)[tmp].vertices_patch[tmp_vertex3->index()].push_back(triangle_position);
// Interface tmp marked as done
itfDoneInThisCycle.insert(tmp);

Since we found the element which shares, with tmp_element, the face, we can interrupt the loop over the elements of the first vertex of the triangle. We do that, by changing the looping variable so that the test expression of the for loop will result false.

k1 = tmp_vertex1->size_element();
}
}
}
}
} // End of the loop over the elements of the first vertex forming the face
} // End of the loop over the element’s faces which might lie on one of the bulk region’s interface
} // End of the loop over the elements of the region whose index is tmp_index

Before moving on considering the next mesh interface, we have to mark all the interfaces that have been processed in this cycle and save the informations we have retrieved about them. We fill the vector itf_idx for each vertex and the vector vert_idx for each interface using the sets we created on purpose itf_per_vert and vert_per_itf, respectively.

// Loop over the interfaces processed in this cycle
for (set<int>::iterator it=itfDoneInThisCycle.begin(); it != itfDoneInThisCycle.end(); it++)
{
    // Interface marked as done
    alreadyDoneItf[*it] = true;
    // Addition of the vertices in the interface list of vertices
    (*Interfaces)[*it].vert_idx.resize(vert_per_itf[*it].size());
copy(vert_per_itf[*it].begin(), vert_per_itf[*it].end(),(*Interfaces)[*it].vert_idx.begin());
} // End of the loop over the interfaces processed in this cycle
} // Closure of the if that checks if the i-th interface has already been considered
} // End of the loop over the mesh interfaces

// Loop over the mesh vertices
for (unsigned int i=0; i < Vertices->size(); i++)
{
    // Addition of the interfaces in the vertices list of interfaces
    (*Vertices)[i].itf_idx.resize(itf_per_vert[i].size());
copy(itf_per_vert[i].begin(), itf_per_vert[i].end(),(*Vertices)[i].itf_idx.begin());
} // End of the loop over the mesh vertices
After we processed the interfaces, we want to deal with the mesh contacts. For each contact we create an object `agr_contact` and we add it to the vector `Contacts`. We read the vector `cElem` and for each contact we extrapolate the elements which constitute it. We extrapolate also the vertices belonging to that contact.

```cpp
Contacts = new vector<agr_contact>;
Contacts->reserve(cElemNumber.size());
start = 0;
end = 0;
// Loop over the contacts
for (unsigned int i=0; i<cElemNumber.size(); i++) {
    // Calculation of the contact index
    size_t idx_ext = size_t (i + Bulks->size());
    // Creation of the object contact
    (*Contacts).push_back(agr_contact(idx_ext, ContactRegions[i].first, this));
}
```

We reserve space for the exact number of elements belonging to the contact. Unfortunately we do not know how many vertices belong to it. Thus we decided to set the maximum number of vertices equal to two times the number of elements of the contact.

```cpp
(*Contacts)[i].surface_elements.reserve(cElemNumber[i]);
(*Contacts)[i].vert_idx.reserve(cElemNumber[i]*2);
start = end;
end = start + cElemNumber[i];
```

We also add the global indices of the vertices which form the contact to its list of triangles `surface_elements`. We also add the local index of the triangle to the list of elements belonging to the vertices patches `vertices_patch`.

```cpp
// Loop over the elements of the contact whose index is idx_ext
for (int j=start; j<end; j++) {
    // Addition of a new triangle in contact list of elements
    (*Contacts)[i].surface_elements.push_back(vector<int>(cElem[j].size()));
    int triangle_position = (*Contacts)[i].surface_elements.size()-1;
    // Loop over the triangle's vertices
    for (unsigned int k=0; k<cElem[j].size(); k++) {
        // Addition of the index of the k-th vertex of the element
        (*Contacts)[i].surface_elements[j-start][k] = cElem[j][k];
        // Addition of the triangle in the vertices patch of elements
        (*Contacts)[i].vertices_patch[cElem[j][k]].push_back(triangle_position);
    }
}
```

We want to establish a connection between the vertex and the contact it belongs to. We cannot simply insert the contact index in the vector `reg_idx` of the vertex because in our loops we may consider multiple times the same vertex. In this way we would have inserted each contact index more than once. To avoid this we use a temporary set to check if the current contact has already been added to the vertex list of regions.

```cpp
temp = Reg_per_Vert[cElem[i][k]].insert(int(idx_ext));
if (temp.second)
    // Addition of the contact in the vertex list of regions (ONLY IF it has not been inserted yet)
    (*Contacts)[i].vert_idx.push_back(cElem[i][k]);
}
```

Now we update the vector `reg_idx` for each vertex using the set we created on purpose `Reg_per_Vert`. 

```cpp
// Sorting of the global indices of the vertices belonging to the i-th contact
sort((*Contacts)[i].vert_idx.begin(),(*Contacts)[i].vert_idx.end());
```

Now we update the vector `reg_idx` for each vertex using the set we created on purpose `Reg_per_Vert`. 

30
for (unsigned int i=0; i<Reg_per_Vert.size(); i++) {
    if(!Reg_per_Vert[i].empty()) {
        (*Vertices)[i].reg_idx.insert((*Vertices)[i].reg_idx.end(),Reg_per_Vert[i].begin(),Reg_per_Vert[i].end());
    }
}

Mesh visualization  Users must be able to display the mesh they are working with. For this purpose we wrote two methods, `saveMesh_hdf5` and `printMesh_xmf`. Their output files make possible to visualize the mesh geometry and topology using a graphic visualization software (for example PARAVIEW).

We decided to use the eXtensible Data Model and Format (XDMF) to store the mesh data. XDMF categorizes data by two main attributes: size and functions. Data can be Light (typically less than a thousand values) or Heavy (megabytes, terabytes, etc.). In addition to raw values, data can refer to Format (rank and dimensions of an array) or Model (how that data is to be used). XDMF uses XML to store Light data and to describe the data Model. Either HDF5 or binary files can be used to store Heavy data. We chose to use an HDF5 file. HDF5 is particularly suitable for huge data storage because it allows to compress them. The data Format is stored redundantly in both the XML and the HDF5 file. This allows tools to parse XML to determine the resources that will be required to access the Heavy data.

XML file  The organization of the XML file begins with the `Xdmf` element. Any element in Xdmf can have a `Name` attribute or a `Reference` attribute. The Name attribute becomes important for grids, while the Reference attribute is used to take advantage of the `XPath` facility. The XPath facility allows to reference specific elements in the document. Xdmf can contain one or more `Domain` elements. A Domain can have one or more `Grid` elements. Each Grid contains a `Topology`, `Geometry` and zero or more `Attribute` elements. Topology specifies the connectivity of the grid while Geometry specifies the location of the grid nodes. Attribute elements are used to specify values such as scalars and vectors that are located at the node, edge, face, cell center or grid center. To specify actual values for connectivity, geometry, and attributes, XDMF defines a `DataItem` element. A DataItem can provide the actual values or provide the physical storage (which is the HDF5 file).

The XML file provided by the method `printMesh_xmf` has one Domain element. The Domain contains a Grid element whose name is Mesh and whose type is Tree. A Tree DataItem is a hierarchical structure of DataItems. The Mesh Grid contains several Grid elements whose type is Collection. A Collection DataItem is a one dimensional array of DataItems. Each Grid element contained in Mesh, except the last, is characterized by a material. Each Grid element identified by a material contains various Grid elements, each one representing a bulk region (made of the material which defines the Grid element). The last Grid element represents the set of contacts. It contains multiple Grid elements, each one representing a contact. There are not actual values in the DataItems, just the position of their storage in the HDF5 file.

HDF5 file  The HDF5 file provided by the method `saveMesh_hdf5` has the following data organization. It contains:

- vertex, a compound dataset consisting of an array $n_v \times 3$ ($n_v$ being the total number of mesh vertices) where the vertices coordinates are stored

- topologies, a group containing the information about the regions (both bulk regions and contacts), within it there are several datasets, one for each region of the mesh:
  - `region_i`, scalar dataset corresponding to the i-th region, it is an array $n_E \times n_e$, where $n_E$ is the total number of elements belonging to the region and $n_e$ is equal to 4 if the region is
a bulk region or \( n_e \) is equal to 3 if the region is a contact, for each element of the region this dataset lists the global indices of the vertices that form the element.

### 1.2.9 Class agr_data

```cpp
class agr_data {
public:
  typedef enum { vertex, edge, element} agr_location;
  agr_data();
  agr_data(agr_mesh * mesh_ext);
  agr_data(const agr_data & orig);
  virtual ~agr_data();
  const double* ReadScalar (agr_location location, string name);
  void WriteScalar (agr_location location, string name, const double* newvalue);
  const double* const* ReadVector(agr_location location, string name);
  void WriteVector (agr_location location, string name, const double* const* newvalue);
  const double* const* ReadTensor(agr_location location, string name);
  void WriteTensor (agr_location location, string name, const double* const* newvalue);
  int saveMeshData_hdf5(string FILE_NAME) const;
  void printMeshData_xmf(string FILE_NAME) const;
  void printBulkData(string FILE_NAME, size_t index) const;
  void printRegionInterfaceData(string FILE_NAME, size_t index) const;
private:
  agr_mesh * mesh;
  map < string, double* > scalar_vertex_datasets;
  map < string, double** > vector_vertex_datasets;
  map < string, double** > tensor_vertex_datasets;
};
```

The class `agr_data` is the class that contains the information about the physical quantities defined on the mesh. The class `agr_data` has four private members and fourteen public members:

- **mesh** it is a pointer to the object mesh (instance of the class `agr_mesh`) on which the physical quantities are defined
- **scalar_vertex_datasets** it is a map that contains the vertex-centered scalar physical quantities; the key of the map is the dataset name, while map value is a native array containing dataset values on mesh vertices
- **vector_vertex_datasets** it is a map that contains the vertex-centered vectorial physical quantities; the key of the map is the dataset name, while map value is a native \( n_v \times d \) array containing dataset values on mesh vertices
- **tensor_vertex_datasets** it is a map that contains the vertex-centered tensorial physical quantities; the key of the map is the dataset name, while map value is a native \( n_v \times n_{components} \) array containing dataset values on mesh vertices, where \( n_{components} = \frac{d(d+1)}{2} \)

- **agr_data()** default constructor, it sets `mesh` equal to a null pointer
- **agr_data(agr_mesh * mesh_ext)** class constructor, it will be discussed later
- **ReadScalar(agr_location location, string name)** this method returns a pointer to the read-only values in the mesh vertices of the scalar physical quantity whose name is `name`, `location` refers to the quantity location type
WriteScalar(agr_location location, string name, const double* newvalue) this method adds to /rewrites in the map scalar_vertex_datasets a scalar physical quantity whose name is name and whose values in the mesh vertices are defined in the array newvalue, location refers to the quantity location type; it is up to the user to ensure that the size of the input array corresponds to the selected location (for example, \(n_v\) for vertex-centered data)

ReadVector(agr_location location, string name) this method returns a pointer to the read-only values in the mesh vertices of the components of the vectorial physical quantity whose name is name, location refers to the quantity location type

WriteVector(agr_location location, string name, const double* const* newvalue) this method adds to the map vector_vertex_datasets a vectorial physical quantity whose name is name and whose values of the components in the mesh vertices are defined in the array newvalue, location refers to the quantity location type; it is up to the user to ensure that the size of the input array corresponds to the selected location (for example, \(n_v \times d\) for vertex-centered data)

ReadTensor(agr_location location, string name) this method returns a pointer to the read-only values in the mesh vertices of the components of the tensorial physical quantity whose name is name, location refers to the quantity location type

WriteTensor (agr_location location, string name, const double* const* newvalue) this method adds to the map tensor_vertex_datasets a tensorial physical quantity whose name is name and whose values of the components in the mesh vertices are defined in the array newvalue, location refers to the quantity location type; it is up to the user to ensure that the size of the input array corresponds to the selected location (for example, \(n_v \times n_{components}\) for vertex-centered data)

saveMeshData_hdf5(string FILE_NAME) this method produces a HDF5 file whose name is FILE_NAME; this file contains the values of the physical quantities defined on the mesh and it can be used (with the output file of the following method) with a graphic visualization software (for example PARAVIEW) to visualize those physical quantities; the method will be discussed in more detail later

printMeshData_xmf(string FILE_NAME) this method produces a XML file whose name is FILE_NAME; this file contains information about the physical quantities defined on the mesh and it can be used with a graphic visualization software (for example PARAVIEW) to visualize those physical quantities; the method will be discussed in more detail later

printBulkData(string FILE_NAME, size_t index) this method produces a VTK file whose name is FILE_NAME; this file contains the values of the physical quantities defined on the bulk region whose index is index, the output file can be used with a graphic visualization software (for example PARAVIEW) to visualize those physical quantities in the index-th bulk region

printRegionInterfaceData(string FILE_NAME, size_t index) this method produces a VTK file whose name is FILE_NAME; this file contains the values of the physical quantities defined on the interface whose index is index, the output file can be used (with the output file of the following method) with a graphic visualization software (for example PARAVIEW) to visualize those physical quantities in the index-th interface

Agr_data class constructor The agr_data class constructor uses the TDR file to retrieve data about the physical quantities defined on the mesh. Unlike the agr_mesh class constructor the data recovered are stored directly in the private members of the class, since they do not require to be processed. We will now describe the main steps of the constructor.
The first thing we do is open the file in read-only mode.

```cpp
H5File file( mesh->file_name(), H5F_ACC_RDONLY );
```

Then we open the outer groups

```cpp
Group collection, geometry;
collection = file.openGroup("collection");
geometry = collection.openGroup("geometry_0");
Group state = geometry.openGroup("state_0");
```

and we read the total number of datasets in the file:

```cpp
Attribute A_tmp;
int nDatasets;
A_tmp = state.openAttribute("number of datasets");
A_tmp.read(PredType::NATIVE_INT32, &nDatasets);
A_tmp.close();
```

Now `nDatasets` contains the number of datasets which is equal to the product of the number of physical quantities for the number of bulk regions. Thus we calculate the number of physical quantities defined on the mesh:

```cpp
int nBulkRegions(mesh->size_bulk());
if ( nDatasets % nBulkRegions != 0) {
    cerr << "WARNING: Something wrong with the number of datasets!" << endl;
    abort();
}
int nDatasetsPerRegion = nDatasets / nBulkRegions;
```

We are ready to read the datasets. We consider only the datasets which are scalars, vectors or symmetric tensors, vertex-centered, and whose values are real numbers. We describe the process which we used to read vertex-centered scalar datasets of type double. The case of scalar datasets of type float is almost identical. Vectors or tensors reading and writing are also very similar. Thus we will show only the part of their processing which differs from the scalars elaboration.

```cpp
H5std_string datasetIdentifier;
int LocType, StructType, ValType, RegionIdentifier, dNumberOfValues(0);
Group G_dataset;
Attribute A_dName, A_dLocType, A_dStructType, A_dValType;
Attribute A_dNumberOfValues, A_dNumberOfRows, A_dRegionIdentifier;
DataSet D_dValues;
double* tmp_double(0);
string tmp_string;
pair< map< string, double* >::iterator, bool > check_insert;
```

// Loop over the physical quantities defined on the mesh
```cpp
for (int i=0; i<nDatasets; i++) {
    datasetIdentifier = "dataset_";
    datasetIdentifier += convertInt( i );
    G_dataset = state.openGroup(datasetIdentifier);
```

We recall that in the attribute `structure type` it is stored the type of the physical quantity (scalar, vector, tensor).

```cpp
A_dStructType = G_dataset.openAttribute("structure type");
A_dStructType.read(PredType::NATIVE_INT32,&StructType);
```

// Switch over the type of physical quantities
```cpp
switch (StructType) {
    // scalar quantity case
    case 0:
```
We recall that the attribute location type is used to distinguish the quantities defined on the vertex from the ones defined on the elements.

```cpp
A_dLocType = G_dataset.openAttribute("location type");
A_dLocType.read(PredType::NATIVE_INT32,&LocType);
// Switch over the physical quantity location
switch (LocType) {
    // vertex-centered case
    case 0:
```

We recall that in the attribute value type it is stored the type of the values of the physical quantity (integer, float, double).

```cpp
A_dValType = G_dataset.openAttribute("value type");
A_dValType.read(PredType::NATIVE_INT32,&ValType);
// Switch over the type of values
switch (ValType) {
    // integer values case
    case 0:
    cerr << " Dataset_" << i << " type is int: this type is not currently supported!" << endl;
    break;
    // float values case
    case 1:
    ...
    break;
    // double values case
    case 2:
```

We read the main attributes of the dataset: the name of the physical quantity and the index of the region to which the dataset refers, and the number of values stored in it (which has to be equal to the number of vertices of the region).

```cpp
A_dName = G_dataset.openAttribute("name");
A_dNumberOfValues = G_dataset.openAttribute("number of values");
A_dRegionIdentifier = G_dataset.openAttribute("region");
A_dName.read(A_dName.getDataType(), tmp_string);
A_dNumberOfValues.read(PredType::NATIVE_INT32,&dNumberOfValues);
A_dRegionIdentifier.read(PredType::NATIVE_INT32,&RegionIdentifier);
A_dName.close();
A_dNumberOfValues.close();
A_dRegionIdentifier.close();
```

Now we read the actual values of the dataset. They represent the values of the scalar quantity in the vertices of the region whose index is RegionIdentifier.

```cpp
tmp_double = new double[dNumberOfValues];
D_dValues = G_dataset.openDataSet("values");
D_dValues.read(tmp_double,PredType::NATIVE_DOUBLE);
D_dValues.close();
```

We remind that each physical quantity is defined with multiple datasets (one for each region where the quantity is defined). Every time we are analysing a dataset, we check if the physical quantity it refers to has already been inserted in the map scalar_real_datasets or not. If it is the first time we read this physical quantity, we have to reserve space for its values. In all the cases we write the dataset values at the position of the map scalar_real_datasets whose key value is the name of the scalar quantity we are considering.

```cpp
check_insert = scalar_real_datasets.insert( pair<string,double*> (tmp_string,0) );
if (check_insert.second) {
```
```cpp
scalar_real_datasets[tmp_string] = new double[nVertices];
for(int ii=0; ii < nVertices; ii++)
    scalar_real_datasets[tmp_string][ii] = 0.0;
}
// Loop over the dataset values
for (int j=0; j<dNumberOfValues; j++)
    // Addition of the dataset values in the map scalar_real_vertex
    scalar_real_datasets[tmp_string][mesh->region(RegionIdentifier)->vertex(j)->index()] = tmp_double[j];
delete[] tmp_double;
break;
// default case: value type not integer, float or double
default: cerr << "Dataset_" << i << " error: Invalid or not considered value type." << endl;
break;
} // End of the switch over the type of values
A_dValType.close();
break;
// default case: the quantity is not vertex-centered
default: cerr << "Dataset_" << i << " is not a quantity defined on the vertex: it is impossible to read!" << endl;
break;
} // End of the switch over the quantity location
A_dLocType.close();
break;
// vectorial quantity case
case 1:
...

The first difference between the scalar case and the vector case is that, in the vector case, we have to read an additional attribute: the number of rows. This attribute represents the vector number of components and it must be equal to the space dimension.

```cpp
A_dNumberOfRows = G_dataset.openAttribute("number of rows");
A_dNumberOfRows.read(PredType::NATIVE_INT32,&dNumberOfRows);
A_dNumberOfRows.close();
if (dNumberOfRows != mesh->dim()) {
    cerr << "ERROR: Vector dimension is supposed to be equal to geometry dimension. Abort." << endl;
    abort();
}
...
```

The second difference between the two case is the way we store the vector data. While a scalar physical quantity is stored in the map `scalar_real_datasets` as a one-dimensional array, a vector quantity is stored as a two-dimensional array \((n_v \times 3)\), where \(n_v\) is the number of mesh vertices) in the map `vector_real_datasets`.

```cpp
...
break;
// tensorial quantity case
case 5:
...
```

In the tensor case the additional attribute we have to read, with respect to the scalar case, is the tensor dimension. This attribute represents the number of elements on the tensor diagonal and it must be equal to the space dimension. All tensors are supposed to be symmetric. Thus, in order to describe a tensor in a 3D space, it is sufficient to store 6 of its 9 components.

```cpp
A_dDimension = G_dataset.openAttribute("dimension");
```
The last difference between the scalar case and the tensor case is the way we store the tensor data. A tensor quantity is stored as a two-dimensional array \((n_v \times 6\), where \(n_v\) is the number of mesh vertices \) in the map `tensor_real_datasets`.

Data visualization The interface we created is meant to be used to perform numerical simulations. The final result of a simulation are physical quantities defined on the mesh vertices. Users must be able to display the values of these quantities. For this purpose we wrote two methods, `saveMeshData_hdf5` and `printMeshData_xmf`. Their output files make possible to visualize the data using a graphic visualization software (for example PARAVIEW).

The method `saveMeshData_hdf5` calls the analogous method of the class `agr_mesh`. Then it modifies the file provided by that method, adding information about the physical quantities stored in the private members of the class `agr_data`.

The method `printMeshData_xmf` does not call its analogous but it writes a new XML file.

XML file The XML file provided by the method `printMeshData_xmf` has one Domain element. The Domain contains a Grid element whose name is Mesh and whose type is Tree. The Mesh Grid contains several Grid elements whose type is Collection. A Collection DataItem is a one dimensional array of DataItems. Each Grid element contained in Mesh, except the last, is characterized by a material. Each Grid element identified by a material contains various Grid elements, each one representing a bulk region (made of the material which defines the Grid element). Within each of these Grid elements there are several DataItems representing the physical quantities defined on the corresponding bulk region. The last Grid element represents the set of contacts. It contains multiple Grid elements, each one representing a contact. Within each of these Grid elements there are several DataItems representing the physical quantities defined on the corresponding contact. There are not actual values in the DataItems, just the position of their storage in the HDF5 file.

HDF5 file The HDF5 file provided by the method `saveMesh_hdf5` is modified by adding:

- **datasets**, a group containing the information about the physical quantities defined on the mesh, within it there are several groups, one for each type of quantity:
  - **scalars**, a group containing the information about the scalar physical quantities defined on the mesh, it contains:
- **vertex_centered**, a group containing the information about the vertex-centered scalar physical quantities defined on the mesh, within it there are several datasets, one for each physical quantity:

- **vectors**, a group containing the information about the vectorial physical quantities defined on the mesh, it contains:
  - **vertex_centered**, a group containing the information about the vertex-centered vectorial physical quantities defined on the mesh, within it there are several datasets, one for each physical quantity:

- **tensors**, a group containing the information about the tensorial physical quantities defined on the mesh, it contains:
  - **vertex_centered**, a group containing the information about the vertex-centered tensorial physical quantities defined on the mesh, within it there are several datasets, one for each physical quantity:

  - **name**, a dataset corresponding to the physical quantity whose name is name, it is an array \( n_v \times n_c \), where \( n_v \) is the total number of vertices in the mesh and \( n_c \) is equal to 1 if the quantity is scalar, \( n_c \) is equal to 3 if the quantity is a vector or \( n_c \) is equal to 6 if the quantity is a tensor, this dataset lists the values of the physical quantity (or of its components) in the mesh vertices

Finally we want to add that we implemented a function `assemble_multiple_xmf` (which is not a method of any class) that assembles information from different time steps, i.e., output files of `printMesh_xmf` or `printData_xmf`, in one XDMF file.
Chapter 2

The Moving Interface Algorithm

After we managed to implement the public interface to store and handle mesh data, we focused on the second goal of our project. During some electronic devices simulations one may have to study physical phenomena where the growth of one material on another occurs. We assume that each region stands for a material, that meaning that a region can not be made of two different materials. Then this phenomenon can be simulated with the movement of one or more interfaces of the region representing the growing material. In order to accomplish this goal we implemented a new method for the class which we called moving_interface.

First we will describe the theoretical method we used to deform three-dimensional grids. Then we will describe how we implemented the algorithm as a class method.

2.1 The ball-vertex method

We consider the problem of deforming unstructured three-dimensional grids. We assume to be in the case when the motion of a portion of the domain is known and one wants to deform the rest of the mesh to accommodate the imposed displacements.

The most commonly used technique for grid deformation is based on the spring analogy method. The basic idea is to create a network of springs connecting all the grid vertices. In its simplest form this method entails that each mesh edge is replaced with a spring, whose stiffness is inversely proportional to the edge length. While this classical method performs reasonably well in a number of cases, it fails as soon as the local grid motion is not small compared to the local mesh size. Furthermore, even if the displacements are small, the edge spring method can not prevent from the creation of nearly flat elements. In fact it can not control the collapse mechanisms of the grid elements. Clearly we want to avoid the elements collapse. Therefore we need a method which can specifically deal with large deformation problem and prevent the elements from collapsing.

A new simple method of controlling collapse mechanisms has been proposed in [3]. This method is based on the idea of complementing the linear edge springs with linear face-vertex springs. These additional springs effectively constrain each vertex within the polyhedral ball that encloses it, contrasting the possible collapse mechanisms of the grid elements. Moreover the presence of the additional springs is also beneficial in terms of mesh quality, since these springs tend to keep each vertex close to the centroid of its ball.

After we have defined the problem to be solved, we will briefly describe the classical edge spring method and then we will explain the ball-vertex method. Finally we will show the incremental algorithm used to actually implement the method.
2.1.1 Notation and definition of the problem

We consider a bounded domain $\Omega \subset \mathbb{R}^3$, and we note its boundary as $\Gamma$. Let $T_h$ be a simplicial triangulation of $\bar{\Omega}$. We will use the capital letters $F$, $E$, $V$ to indicate faces, edges and vertices, respectively. A generic topological entity belonging to one of these four types is labelled $T$. A generic tetrahedron is indicated with the symbol $K$. List of entities are indicate as $\{\cdot\}$, $\{T\}_S$ is used to indicate a list of entities in the set $S$.

We consider a problem where we are interested in deforming the domain $\Omega$, and hence the grid $T_h$ associated with it, in order to accommodate some given displacement on a portion of the domain. The basic ideas behind this method is to define suitable fictitious structural properties for the domain. Then the deformed configuration of the elastic domain can be computed, under the action of the driving displacements.

The fictitious elastic problem used to compute a deformed grid configuration can be formulated in different ways. First of all, the problem can be regarded as transient or steady. Since the grid deformation problem is completely artificial and no physical solution field is associated with it, it is usually convenient to consider the steady version of the formulation. This is true even when the simulated physical phenomenon is transient. Furthermore the fictitious structural model can be either discrete or continuous. We consider the discrete case, where the fictitious problem is obtained by defining a suitable network of springs associated with the grid $T_h$. The problem becomes how to construct the best possible network of springs that:

- is simple and inexpensive to compute
- does not contain collapse mechanisms
- leads to graded and well shaped deformed grid, even for large imposed displacements.

The domain can be partitioned according to the following criterion: $\Omega = \Omega_M \cup \Omega_I \cup \Omega_0$. The moving part of the domain is noted $\Omega_M$, and its corresponding discrete version in $T_h$ is $\Omega_{h,M}$. On this domain portion the displacements are known. Or rather the displacement of the grid vertices in $\Omega_{h,M}$ are known, i.e.

$$u_i = g_i \quad \forall i \in \{V\}_{\Omega_{h,M}}. \tag{2.1}$$

The portion of the domain where no driving displacements are imposed can be further subdivided into two parts, corresponding to the terms $\Omega_I$ and $\Omega_0$. On $\Omega_{h,0}$, the corresponding discrete version in $T_h$ of $\Omega_0$, the grid displacements are requested to be null:

$$u_i = 0 \quad \forall i \in \{V\}_{\Omega_{h,0}}. \tag{2.2}$$

Finally we denote by $\Omega_I$ the set of vertices which has to be moved to accommodate the given displacements. As regards the boundary $\Gamma$ we impose that the displacements of all its vertices are null.

2.1.2 The basic edge spring method

Given two vertices, $i$ and $j$, the edge-vector from $i$ to $j$ is defined as

$$e_{ij} = x_j - x_i$$

and its length is

$$L_{ij} = \sqrt{e_{ij} \cdot e_{ij}}.$$ 

The unit edge-vector can then be written as

$$i_{ij} = \frac{e_{ij}}{L_{ij}}.$$
The displacements of vertices \( i \) and \( j \) are noted \( u_i \) and \( u_j \), respectively. The stretching on the edge spring is computed as \( (u_i - u_j) \cdot i_{ij} \). The resulting force on the vertex \( i \) is aligned along the unit vector \( i_{ij} \) and can be written as

\[
\mathbf{f}_{ij}^{\text{edge}} = k_{ij} (u_i - u_j) \cdot i_{ij} = -\mathbf{f}_{ji}^{\text{edge}} \tag{2.3}
\]

where \( k_{ij} \) is the spring stiffness. The spring stiffness is typically chosen as inversely proportional to the edge length:

\[
k_{ij} = \frac{1}{L_{ij}}
\]

so that short edges are stiffer than longer ones, which provides a beneficial effect in the control of the local element deformation.

The position of each vertex belonging to \( \Omega_{h,I} \), the corresponding discrete version in \( T_h \) of \( \Omega_f \) is found by writing its equilibrium under the effect of all its \( n_{E} \) edge-connected springs:

\[
\sum_{j=1}^{n_{E}} \mathbf{f}_{ij}^{\text{edge}} = 0. \tag{2.4}
\]

As a result we obtain a linear system of equations which can be solved in a variety of ways. Some of them, including the one we chose, require the assembly of the stiffness matrix. The four \( 3 \times 3 \) block entry contributions to the stiffness matrix due to the edge connecting vertices \( i \) and \( j \) are readily found by inspection of Eq. 2.3:

\[
\mathbf{K}_{ii} = -k_{ij} i_{ij} i_{ij}^T \quad \mathbf{K}_{ij} = \mathbf{K}_{ji} = k_{ij} i_{ij} i_{ij}^T \quad \mathbf{K}_{jj} = -k_{ij} i_{ij} i_{ij}^T. \tag{2.5}
\]

### 2.1.3 The ball-vertex spring method

The basic idea of this method is to introduce additional linear springs which resist the motion of a mesh vertex towards each one of its region opposed faces. For each vertex \( i \) we call \( P_i \) its patch of elements. For each face \( F_i \) opposite to \( i \) in \( P_i \) a linear spring is constructed connecting \( i \) with its projection \( p \) on the plane of the face. This additional spring is exemplified in Fig. 2.1 for a single tetrahedron connected to a vertex, for clarity. Once the additional springs are constructed for each tetrahedra belonging to \( P_i \), one has effectively constrained the vertex \( i \) not to leave the polyhedral ball \( B_i \) that encloses it.

![Figure 2.1: Ball-vertex spring method: additional linear spring connecting the vertex i with the opposite face in a tetrahedron.](image)
The position of each vertex belonging to \( \Omega_{p,i} \) is found by writing its equilibrium under the combined effect of its edge-connected springs, together with the additional ball-vertex springs. If \( u_i \) and \( u_p \) indicate the displacements of vertex \( i \) and of its projection \( p \) on the face \( F_i \), respectively, the resulting force on \( i \) can be expressed as

\[
f_{ip}^{\text{face-vertex}} = k_{ip} (u_p - u_i) \cdot i_{ip} = -f_{pi}^{\text{face-vertex}}
\]

(2.6)

where, exactly as before, \( e_{ip} = x_p - x_i \), \( L_{ip} = \sqrt{e_{ip} \cdot e_{ip}} \), \( i_{ip} = e_{ip} / L_{ip} \). The spring stiffness is chosen here again according to the same criterion \( k_{ip} = \frac{1}{L_{ip}} \). The only significant difference with respect to the edge spring case, is given by the fact that \( u_p \) is now the displacement of a virtual point, and not of an existing vertex. A simple way to address this issue is to calculate the displacement of \( p \) by interpolating the displacements of the three face vertices, \( j, k \), and \( l \). First, \( x_p \) is computed as the normal projection of the vertex \( i \) on the face \( F_i \) as

\[
x_p = x_i - (x_i - x_j) \cdot n 
\]

(2.7)

where \( n \) is the unit normal to \( F_i \):

\[
n = \frac{i_{jk} \times i_{jl}}{\| i_{jk} \times i_{jl} \|}.
\]

(2.8)

Given \( x_p \), the interpolation coefficients \( \xi \) and \( \eta \) corresponding to \( p \) can be computed such that

\[
x_p = \xi x_j + \eta x_k + (1 - \xi - \eta) x_l.
\]

Even if \( p \) falls outside the face \( F_i \) no special actions are required. Solving \( \xi \) and \( \eta \), one finds

\[
\xi = \frac{(x_j - x_i) (x_p - x_i)}{\| x_j - x_i \|^2}
\]

\[
\eta = \frac{(x_k - x_i) (x_p - x_i)}{\| x_k - x_l \|^2}.
\]

Finally, given \( \xi \) and \( \eta \), the interpolated displacement of \( p \) is obtained as

\[
u_p = \xi u_j + \eta u_k + (1 - \xi - \eta) u_l.
\]

(2.9)

In a similar manner, the spring force applied to point \( p \) is linearly distributed among the vertices \( j, k \) and \( l \) of the ball face \( F_i \). The virtual work between \( i \) and \( p \) due to the infinitesimal variation of its two end points is

\[
\partial W_{ip} = -f_{ip}^{\text{face-vertex}} \cdot \partial u_i - f_{ip}^{\text{face-vertex}} \cdot \partial u_p
\]

(2.10)

where \( \partial u_p \) is the virtual variation of \( x_p \). Accordingly, the forces applied by the spring on the vertices \( j, k \) and \( l \) of the ball face \( F_i \) are

\[
f_{ip,j}^{\text{face-vertex}} = \xi f_{ip}^{\text{face-vertex}}
\]

\[
f_{ip,k}^{\text{face-vertex}} = \eta f_{ip}^{\text{face-vertex}}
\]

\[
f_{ip,l}^{\text{face-vertex}} = (1 - \xi - \eta) f_{ip}^{\text{face-vertex}}.
\]

Even in the case of the ball-vertex springs, the resulting system of linear equations can be solved in different ways. The local stiffness matrix \( K_{ip} \), due to the insertion of one spring is readily found, by using the principle of virtual work (Eq. 2.10), as

\[
K_{ip} = -k_{ip} \begin{bmatrix}
    i_{ip}^T & -\xi i_{ip}^T & -\eta i_{ip}^T & -(1 - \xi - \eta) i_{ip}^T \\
    \xi^2 i_{ip}^T & \xi \eta i_{ip}^T & \xi (1 - \xi - \eta) i_{ip}^T & \eta (1 - \xi - \eta) i_{ip}^T \\
    \eta^2 i_{ip}^T & \eta (1 - \xi - \eta) i_{ip}^T & \eta (1 - \xi - \eta)^2 i_{ip}^T & \text{symm.}
\end{bmatrix}
\]

(2.11)
2.1.4 The incremental displacement algorithm

The implementation of the algorithm we used is

\[
\begin{align*}
\forall i & \in \{ V \}_{T_h} \\
\alpha^r & = x_i^{\text{orig}} \quad r = 0 \\
\text{do} & \\
& \text{compute spring stiffness at current configuration } x_i^r \left( x_i^0 = x_i^{\text{orig}} \right) \forall i \in \{ V \}_{T_h} \\
& \text{compute increment coefficient } \alpha^r \text{ at } x_i^r \forall i \in \{ V \}_{\Omega_{h,M}} \text{ [current imposed displacements]} \\
& u_i^r = \alpha^r g_i, \forall i \in \{ V \}_{\Omega_{h,M}} \text{ [current imposed displacements]} \\
& \text{Solve linear problem and compute vertex displacements } u_i^r \forall i \in \{ V \}_{\Omega_{h,L}} \\
& x_i^{r+1} = x_i^{\text{orig}} + u_i^r \forall i \in \{ V \}_{T_h \setminus \Omega_{h,L}} \text{ [update vertex positions]} \\
& r = r + 1 \\
\text{while } & \alpha^r < 1
\end{align*}
\]

The algorithm operates as follows. The initial positions of the grid vertices are stored for later use. Next, an iteration started, with iteration index \( r \). All lumped parameters which are necessary for constructing the edge springs, the additional ball-vertex springs and their associated data are computed as functions of the current configuration \( x_i^r \forall i \in \{ V \}_{T_h} \), where \( x_i^0 = x_i^{\text{orig}} \). The displacement scaling factor \( \alpha^r \leq 1 \) is computed. It is used to compute the displacement increment for each moving vertex that will be applied at the current iteration. This is given as

\[
u_i^r = \alpha^r g_i, \quad \forall i \in \{ V \}_{\Omega_{h,M}}.
\]

(2.12)

In this way if the given displacement is too large with respect to the local grid size, just a smaller part of it is imposed. The scaling factor \( \alpha^r \) is computed in the following way. For each moving vertex \( i \) belonging to \( \Omega_{h,M} \), the list of faces \( B_i \) is computed. The minimum distance \( d_i^r \) between vertex \( i \) and each face \( F_i \in B_i \) is then calculated as

\[
d_i^r = \min_{F_i \in B_i} \text{dist} \left( x_i^r, F_i \right).
\]

Based on \( d_i^r \), a scaling factor for the applied displacement at vertex \( i \) at the current iteration can be computed as

\[
\alpha_i^r = \left( \frac{d_i^r}{s} - u_i^{r-1} \right) \frac{1}{g_i}
\]

(2.13)

where \( s \) is a safety factor, typically chosen as \( s = 2 \), \( u_i^{r-1} = \sqrt{\mathbf{u}_i^{r-1} \cdot \mathbf{u}_i^{r-1}} \) is the magnitude of the applied displacement at the previous iteration, and \( g_i = \sqrt{g_i \cdot g_i} \) is the magnitude of the desired full applied displacement. In practice, \( \alpha_i^r \) represents the safe scaling factor for the vertex \( i \) at the current iteration which ensures that the corresponding incremental displacement will not locally entangle the grid. Finally, the global scaling factor, \( \alpha^r \), is chosen as the minimum of all scaling coefficients computed for the various moving vertices, or it is set to one once the full required displacement has been reached:

\[
\alpha^r = \min \left( 1, \min_{i \in \{ V \}_{\Omega_{h,M}}} \alpha_i^r \right).
\]

Based on the computed value \( \alpha^r \), the imposed displacements at iteration \( r \) are evaluated according to Eq. 2.12. At this point, the linear problem resulting from the spring analogy method is solved. The solution are the displacements \( u_i^r \) for all vertices \( i \) belonging to \( \Omega_{h,L} \). The increment iteration is completed by updating the original positions of all the grid vertices, except those belonging to \( \Omega_{h,0} \). Iterations are terminated when \( \alpha^r \) is equal to 1, i.e. when the full displacements have been applied.

2.2 The moving_interface method

In order to perform the movement of one or more interfaces of the mesh we created a method in the class \texttt{agr_mesh}. Here is its definition:
int moving_interface(vector<int> const & itflist, vector<vector<vector<double> > > const & itfdisplacement, double tol_pref);

The method returns the number of cycles \( r \) performed by the moving algorithm. It has three input arguments: the indices list of the mesh interfaces to be moved \( \text{itflist} \); the list, for each interface in \( \text{itflist} \), of the imposed displacement to its vertices \( \text{itfdisplacement} \); the tolerance to be used to select the grid vertices which must be moved to accommodate the given displacements \( \text{tol}\_\text{pref} \). We will now describe the body of method.

First the coordinates of the mesh vertices are stored as initial positions to which the computed displacements will be added. They are stored in a \( \text{vector}\<\text{vector}\<\text{double}\>\> \) called \( \text{OrigCoord} \) whose size is equal to the number of mesh vertices. Then we split the mesh vertices in three groups by labeling each vertex with an integer number representing one of the group. Number 1 represents the vertices belonging to the interfaces which have to be moved (\( \Omega_{h,M} \)), their displacements are known. Number 2 represents the vertices whose displacements have to be computed (\( \Omega_{h,I} \)). And number 0 represents the vertices whose displacements will be imposed null (\( \Omega_{h,0} \)). The labels are stored in a vector of integers (\( \text{vector}\<\text{short int}\> \)) called \( \text{vertex\_groups} \). It has the size of the total number of mesh vertices, and all its values are initialized to zero. Identifying the vertices belonging to group 1 is easy. In fact it is sufficient to retrieve the vertices of each interface in \( \text{itflist} \). On the other hand, the process is more complex for group 2. We check each mesh vertex, it will belong to groups 2 if

- it does not belong to the mesh outer borders,
- it is sufficiently close to one of the moving interfaces.

We assume our grid to be a parallelepiped, and we identify the maximum and minimum value for each spatial coordinate of the mesh vertices. We store this information as \text{double} objects in \( \text{ABSmin}_x, \text{ABSmax}_x, \text{ABSmin}_y, \text{ABSmax}_y, \text{ABSmin}_z, \text{ABSmax}_z \), with obvious meaning. We do the same thing for the vertices belonging to the interfaces in \( \text{itflist} \). This information is stored in \( \text{vector}\<\text{double}\> \) objects, each component representing an interface of \( \text{itflist} \), called \( \text{ITFmin}_x, \text{ITFmax}_x, \text{ITFmin}_y, \text{ITFmax}_y, \text{ITFmin}_z, \text{ITFmax}_z \). We calculate a tolerance \( \text{toll} \) equal to the product of \( \text{tol}\_\text{pref} \) by the minimum grid step (i.e., the smallest edge length). We consider a vertex belonging to the outer borders if one of its coordinates is equal to the corresponding absolute maximum or minimum value \( \text{ABSmax}/\text{ABSmin} \). We consider a vertex sufficiently close to a moving interface if its coordinates values are in the range \([\text{ITFmin} - \text{toll}, \text{ITFmax} + \text{toll}] \).

Once we have identified the three vertices groups, we have to compute the displacements of the vertices belonging to group 2. To do that we have to build the linear system of equations we shown in the previous section. We create two \text{map} objects \( \text{Kmovements} \) and \( \text{Omovements} \) to contain the full given displacements and the partial displacements imposed in the previous iteration of the incremental algorithm, respectively. Their keys are the indexes of the vertices belonging to group 1. Values in \( \text{Omovements} \) are initially set to zero. In \text{known\_moves} we store the number of vertices belonging to group 1. To handle the linear system we used the ltest version of the Eigen library (3.1.3 release).

```cpp
size_t nV = this->size_vertex();
double alpha_r(0);
int count_k(0);
vector<double> alphas(known_moves);
typedef Eigen::Triplet<double> T;
vector<T> triplet_list;
int r = 0;
```

We start the loop over the algorithm iterations. At each iteration \( r \) we build a linear system whose dimensions are \( 3nV \times 3nV \), where \( nV \) is the total number of mesh vertices. The factor 3 is due to the fact that we have to solve the system with respect to the displacements which are vectors of three components. We do that because we decided to solve the global system for all the grid vertices. For vertex belonging to group 0 we will impose eq. 2.2. For vertex belonging to group 1 we will impose
eq. 2.12. For vertex belonging to group 2 we will write its equilibrium equation, considering the effect of both the edge springs and the ball-vertex springs.

// Loop over the algorithm iteration
while (alpha_r < 1) {
    // Creation of the global matrix
    Eigen::SparseMatrix<double> A(3*nV,3*nV);
    // Creation of the global load vector
    Eigen::VectorXd b = Eigen::VectorXd::Zero(3*nV);
    
    We create a map mat_elem to contain the elements of the matrix, its keys being the couples row index-column index.

    map<pair<unsigned int,unsigned int>,double> mat_elem;
    
    First we fill the matrix with the block entry contributions due to the edge springs (see 2.5 ). We compute the edge spring stiffness and the unit edge-vector using the formula shown in the previous section.

    // Loop over the edges of the mesh
    for (unsigned int i=0; i<this->size_edge(); i++) {
        // Recover of the edge endpoints
        agr_vertex* v1 = this->edge(i)->start();
        size_t id_v1 = v1->index();
        agr_vertex* v2 = this->edge(i)->end();
        size_t id_v2 = v2->index();
        // Calculation of the edge spring stiffness
        double stiffness = 1/this->edge(i)->length();
        // Calculation of the unit edge-vector
        vector<double> edge_versor(3);
        for (unsigned int j=0; j<edge_versor.size(); j++)
            edge_versor[j] = (v1->coord()[j]-v2->coord()[j])*stiffness;
        
        We insert the elements only in the rows corresponding to the indices of the vertices belonging to group 2.

        if (vertex_groups[id_v1]==2) {
            for (unsigned int j=0; j<edge_versor.size(); j++) {
                for (unsigned int k=0; k<edge_versor.size(); k++) {
                    mat_elem[make_pair(3*id_v1+j,3*id_v1+k)] += -stiffness*edge_versor[j]*edge_versor[k];
                    mat_elem[make_pair(3*id_v1+j,3*id_v2+k)] += stiffness*edge_versor[j]*edge_versor[k];
                }
            }
        }
        
        if (vertex_groups[id_v2]==2) {
            for (unsigned int j=0; j<edge_versor.size(); j++) {
                for (unsigned int k=0; k<edge_versor.size(); k++) {
                    mat_elem[make_pair(3*id_v2+j,3*id_v2+k)] += -stiffness*edge_versor[j]*edge_versor[k];
                    mat_elem[make_pair(3*id_v2+j,3*id_v1+k)] += stiffness*edge_versor[j]*edge_versor[k];
                }
            }
        }
    }
    // End of the loop over the edges of the mesh
    
    Now we fill the matrix with contributions due to the ball-vertex springs (see 2.11 ). While we are doing this we profit by the cycles to calculate the scaling factors $\alpha_i^r$.

    // Loop over the vertices of the mesh
    for (unsigned int i=0; i<nV; i++) {
        agr_vertex* v = this->vertex(i);
If the vertex we are considering belongs to group 0 or 1, its corresponding rows will have all the off-diagonal elements null and the diagonal elements equal to 1.

```cpp
if (vertex_groups[i]!=2) {
    for (int j=0; j<dimension; j++)
        mat_elem[make_pair(3*i+j,3*i+j)] = 1;
}
```

We consider all the elements to which the vertex we are examining belongs. For each element we identify the three vertices \((j, k \text{ and } l)\) forming the face opposed to the vertex we are inspecting.

```cpp
if (vertex_groups[id_v[0]]==2 || vertex_groups[id_v[1]]==2 || vertex_groups[id_v[2]]==2 || vertex_groups[id_v[3]]==2 ) {
    vector<double> edge1(3),edge2(3), normal_versor(3), vp(3);
    agr_vertex* vj = this->vertex(id_v[1]);
    agr_vertex* vk = this->vertex(id_v[2]);
    agr_vertex* vl = this->vertex(id_v[3]);
    // Calculation of the vectors of the directions of the edges connecting j to k and k to l
    for (unsigned int k=0; k<edge1.size(); k++)
        edge1[k] = (vj->coord()[k]-vk->coord()[k]);
    for (unsigned int k=0; k<edge2.size(); k++)
        edge2[k] = (vj->coord()[k]-vl->coord()[k]);
    // Calculation of the unit normal to \(F_i\)
    normal_versor[0] = edge1[1]*edge2[2]-edge1[2]*edge2[1];
    normal_versor[1] = -edge2[2]*edge1[0] + edge1[2]*edge2[0];
    normal_versor[2] = edge1[0]*edge2[1]-edge1[1]*edge2[0];
    double norm = sqrt(pow(normal_versor[0],2)+pow(normal_versor[1],2)+pow(normal_versor[2],2));
    for (unsigned int k=0; k<edge1.size(); k++)
        normal_versor[k] = normal_versor[k]/norm;
}
```

Now we can compute \(x_p\) using 2.7.
for (unsigned int k=0; k<edge1.size(); k++)
vp[k] = v->coord()[k]-dot_prod*normal_versor[k];

Then we compute the ball-vertex spring stiffness and the unit spring-vector using the formula shown in the previous section. We profit by the calculation of the spring stiffness to compute the distance between the vertex \(i\) and the face \(F_i\). In fact such distance is equal to the spring length which corresponds to the inverse of the stiffness. Since actually we have to compute the minimum distance between the vertex \(i\) and its opposed faces \(F_i\) belonging to the ball \(B_i\), we stored the information about the spring length only if it is smaller than the minimum distance computed so far.

// Calculation of the ball-vertex spring stiffness
double stiffness = 1/sqrt(pow(v->coord()[0]-vp[0],2)+pow(v->coord()[1]-vp[1],2)+pow(v->coord()[2]-vp[2],2));
if (1/stiffness<dist)
// Calculation of distance between the vertex \(i\) and the face \(F_i\),
dist = 1/stiffness;

// Calculation of the unit spring-vector
vector<double> spring_versor(3);
for (unsigned int k=0; k<spring_versor.size(); k++)
spring_versor[k] = (v->coord()[k]-vp[k])*stiffness;

We are now ready to calculate the interpolation coefficients \(\xi\) and \(\eta\) using the formula given in the previous section.

double csi(0), eta(0);
// Calculation of the coefficients \(\xi\)
for (unsigned int k=0; k<edge2.size(); k++)
csi += edge2[k]*(vp[k]-vl->coord()[k]);
csi /= pow(edge2[0],2)+pow(edge2[1],2)+pow(edge2[2],2);
// Calculation of the coefficients \(\eta\)
for (unsigned int k=0; k<edge2.size(); k++)
eta += (vk->coord()[k]-vl->coord()[k])*(vp[k]-vl->coord()[k]);
eta /= pow(vk->coord()[0]-vl->coord()[0],2)+pow(vk->coord()[1]-vl->coord()[1],2)+pow(vk->coord()[2]-vl->coord()[2],2);

We insert the elements only in the rows corresponding to the indexes of the vertices belonging to group 2.

if (vertex_groups[id_v[0]]==2) {
for (unsigned int k=0; k<spring_versor.size(); k++) {
for (unsigned int p=0; p<spring_versor.size(); p++) {
mat_elem[make_pair(3*id_v[0]+k,3*id_v[0]+p)] += -stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[0]+k,3*id_v[1]+p)] += csi*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[0]+k,3*id_v[2]+p)] += eta*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[0]+k,3*id_v[3]+p)] += (1-eta-csi)*stiffness*spring_versor[k]*spring_versor[p];
}
}
}

if (vertex_groups[id_v[1]]==2) {
for (unsigned int k=0; k<spring_versor.size(); k++) {
for (unsigned int p=0; p<spring_versor.size(); p++) {
mat_elem[make_pair(3*id_v[1]+k,3*id_v[1]+p)] += -pow(csi,2)*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[1]+k,3*id_v[0]+p)] += csi*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[1]+k,3*id_v[2]+p)] += -csi*eta*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[1]+k,3*id_v[3]+p)] += -csi*(1-eta-csi)*stiffness*spring_versor[k]*spring_versor[p];
}
}
}

if (vertex_groups[id_v[2]]==2) {
for (unsigned int k=0; k<spring_versor.size(); k++) {
for (unsigned int p=0; p<spring_versor.size(); p++) {
mat_elem[make_pair(3*id_v[2]+k,3*id_v[2]+p)] += -pow(eta,2)*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[2]+k,3*id_v[1]+p)] += eta*stiffness*spring_versor[k]*spring_versor[p];
mat_elem[make_pair(3*id_v[2]+k,3*id_v[0]+p)] += -csi*eta*stiffness*spring_versor[k]*spring_versor[p];
}
}
}

47
for (unsigned int p = 0; p < spring_versor.size(); p++) {
    mat_elem[make_pair(3*id_v[2]+k,3*id_v[2]+p)] += -pow(eta,2)*stiffness*spring_versor[k]*spring_versor[p];
    mat_elem[make_pair(3*id_v[2]+k,3*id_v[0]+p)] += eta*stiffness*spring_versor[k]*spring_versor[p];
    mat_elem[make_pair(3*id_v[2]+k,3*id_v[1]+p)] += -csi*eta*stiffness*spring_versor[k]*spring_versor[p];
}
}
}
}

if (vertex_groups[id_v[3]]==2) {
    for (unsigned int k = 0; k < spring_versor.size(); k++) {
        for (unsigned int p = 0; p < spring_versor.size(); p++) {
            mat_elem[make_pair(3*id_v[3]+k,3*id_v[3]+p)] += -pow(1-eta-csi,2)*stiffness*spring_versor[k]*spring_versor[p];
            mat_elem[make_pair(3*id_v[3]+k,3*id_v[0]+p)] += (1-eta-csi)*stiffness*spring_versor[k]*spring_versor[p];
            mat_elem[make_pair(3*id_v[3]+k,3*id_v[1]+p)] += -csi*(1-eta-csi)*stiffness*spring_versor[k]*spring_versor[p];
        }
    }
}
}  // End of the if which checks if at least one of the element vertices belongs to group 2
}  // End of the loop over the elements to which the i-th mesh vertex belongs

After computing the global stiffness matrix elements, we calculate the scaling factor $\alpha^i_r$ of the vertex $i$ (if the vertex belongs to group 1) using \ref{eq:2.13}.

if (vertex_groups[i]==1) {
    double old_movem = sqrt(pow(Omovements[i][0],2)+pow(Omovements[i][1],2)+pow(Omovements[i][2],2));
    double full_movem = sqrt(pow(Kmovements[i][0],2)+pow(Kmovements[i][1],2)+pow(Kmovements[i][2],2));
    alphas[count_k] = (dist/2 + old_movem)/full_movem;
    count_k++;
}  // End of the loop over the vertices of the mesh

Since we have computed all the global matrix elements, we can now assemble the matrix. We use an Eigen sparse matrix object to store it, and in order to fill the sparse matrix we exploit the Eigen method setFromTriplets. This method builds a sparse matrix given a list of triplets. A triplet is a tuple (row ind, column ind, value) defining a non-zero element of the matrix. The list of triplets must contain the matrix elements. They can be in any order, they may even be duplicated and, in this case, the method will sum up the duplicate items.

// Filling of the triplets list
triplet_list.reserve(mat_elem.size());
map<pair<unsigned int,unsigned int>,double>::iterator it
for (it = mat_elem.begin(); it != mat_elem.end(); it++)
    triplet_list.push_back(T(it->first.first,it->first.second,it->second));
// Cleaning if the map
mat_elem.clear();
// Filling of the global stiffness matrix
A.setFromTriplets(triplet_list.begin(),triplet_list.end());

Now we want to fill the global load vector. First we calculate the scaling factor $\alpha^r$ using the formula shown in the previous section.

// Calculation of the scaling factor
alpha_r = *min_element(alphas.begin(),alphas.end());
if (alpha_r > 1)
    alpha_r = 1;
Then we fill the load vector. At the rows corresponding to vertices belonging to group 1 we write the
given displacement scaled with the factor $\alpha r$. All other elements of the vectors are null, either because
we are imposing the vertices displacements null or because we are imposing the forces equilibrium on
the vertices.

```cpp
// Loop over the mesh vertices
for (unsigned int i=0; i<nV; i++) {
    if (vertex_groups[i]==1) {
        for (unsigned int j=0; j<Kmovements[i].size(); j++)
            b(3*i+j) = alpha_r*Kmovements[i][j];
    }
    else {
        for (unsigned int j=0; j<Kmovements[i].size(); j++)
            b(3*i+j) = 0.0;
    }
} // End of the loop over the mesh vertices
```

We are now ready to solve the linear system. We decided to use a GMRES (Generalized Minimal
RESidual method) solver which is a general solver for sparse square problems. As preconditioner we
use the default diagonal preconditioner, and we set the tolerance for the method to $10^{-10}$. If there are
any problems with the preconditioner computation or with the solution of the linear system, an error
message is printed and the the program is stopped.

```cpp
// Definition of the solver
Eigen::GMRES<Eigen::SparseMatrix<double> > solver;
// Computation of the preconditioner
solver.compute(A);
if (solver.info()!=Eigen::Success) {
    cerr<<" Error in the matrix!!! Solving failure! Abort! "<<endl;
    abort();
}
// Setting of the method tolerance
solver.setTolerance(1e-10);
// Solution of the linear system
Eigen::VectorXd sol = solver.solve(b);
if (solver.info()!=Eigen::Success) {
    cerr<<" Error in the matrix!!! Solving failure! Abort! "<<endl;
    abort();
}
```

Finally we update the vector Omovements with the last scaled imposed displacements, and we also
update the vertices coordinates with the newly computed displacements. Furthermore we increment
the algorithm iteration $r$.

```cpp
// Loop over the mesh vertices
for (unsigned int i=0; i<nV; i++) {
    // Loop over the i-th mesh vertex coordinates
    for (int j=0; j<dimension; j++) {
        if ( vertex_groups[i] == 1 )
            Omovements[i][j] = b(3*i+j);
        (*Vertices)[i].coordinates[j] = OrigCoord[i][j] + b(3*i+j);
    }
    if ( vertex_groups[i] == 2 )
        (*Vertices)[i].coordinates[j] = OrigCoord[i][j] + sol(3*i+j);
} // End of the loop over the i-th mesh vertex coordinates
// Cleaning of the triplets list vector
```
triplet_list.clear();
count_k = 0;
// Incrementation of the cycle iteration
r++;
} // End of the loop over the algorithm iterations

Lastly we return the total number of iterations $r$.

### 2.3 Simulation results

In this section we will show the outcomes of the simulations regarding the moving interface algorithm. First we want to prove the goodness of the implemented method. Then we will show two test cases as examples of possible applications.

#### 2.3.1 Numerical validation

In order to prove the goodness of the method `moving_interface` we applied it to a rather simple structure, a parallelepiped with six regions inside. Figure 2.2 shows the lower part of the parallelepiped, the sixth region, which can not be seen in the figure, is located above and has a height equal to 1.

![Figure 2.2](image)

We decided to focus on the inner parallelepiped (red region), and in particular to move the interface representing its upper face. We applied a periodical displacement to all the vertices of that interface. In order to estimate the quality of the resulting grids we used geometric criteria which give an objective assessment. We consider quality measures based on the ratio of the radii of the inscribed and circumscribed spheres to each element, noted $\bar{r}$ and $R$, respectively. This gives a measure of the stretching of the elements. We also consider quality measures based on dihedral angles of each element. Large dihedral angles can negatively impact the solution of partial differential equations, affecting the discretization error and the conditioning of the discrete problem. Very small dihedral angles indicate the generation of sliver elements, which should also be avoided. The smallest dihedral angle between two neighboring faces in a grid element is defined as

$$\gamma_{\text{min}} = \min_{K \in \mathcal{T}_h} \min_{E \in \{E\}_K} \gamma_{E,K}$$

where $\{E\}_K$ is the the list of edges bounding simplex $K$, while the largest dihedral angle is

$$\gamma_{\text{max}} = \max_{K \in \mathcal{T}_h} \max_{E \in \{E\}_K} \gamma_{E,K}.$$
The dihedral angle at edge $E$ formed by faces $F_1$ and $F_2$ of simplex $K$ is $\gamma_{E,K} = \pi \pm \arccos (\mathbf{n}_{F_1} \cdot \mathbf{n}_{F_2})$, being $\mathbf{n}_{F_i}$ the normal to face $F_i$. Similarly, we define the average small and large dihedral angles in the grid as

$$\gamma_{\text{min},\text{avg}} = \frac{1}{n_K} \sum_{K \in T_h} \min_{E \in \{E\}_K} \gamma_{E,K},$$

$$\gamma_{\text{max},\text{avg}} = \frac{1}{n_K} \sum_{K \in T_h} \max_{E \in \{E\}_K} \gamma_{E,K},$$

respectively, where $n_K$ is the number of tetrahedra in the grid.

We will briefly explain the salient parts of the main file we used to run this test. We create the vector itfdisplacement to contain the movement we want the vertices to perform.

```cpp
vector<vector<vector<double>>> itfdisplacement(1);
itfdisplacement[0].resize(mesh->regioninterface(itflist[0])->size_vertex());
```

We compute the maximum and minimum and the average values of the $y$-coordinates of the moving interface vertices and we store them in $y_{\text{max}}$, $y_{\text{min}}$ and $y_{\text{c}}$, respectively. Now we have to define the vertices displacements. For the sake of simplicity we decide to move the vertices only in the vertical direction ($z$ axis). The displacement of each vertex is time dependent. It is also space dependent but it actually depends only on the $y$–coordinate of the vertex. We want the new position of the grid vertex $i$ to be equal to:

$$x_i^t = \pi \sin \left( \frac{2\pi t}{T} \right) \left( 1 + \sin \left( \frac{2\pi \left( x_i^t - y_C \right)}{y_{\text{max}} - y_{\text{min}}} \right) \right),$$

where $\pi$ is a given displacement and $T$ is the period. We divide the period in $N$ steps ($N = 100$), and we decide to perform four periodical cycles, finally $n$ denotes the current time step.

```cpp
int N(100), nCycles(4), r(0);
unsigned int nE = mesh->size_element();
double max_displ = 0.25;
vector<double> timesteps;
// Loop over time
for (int n=0; n < N*nCycles+1; n++) {
    // Loop over the grid vertices
    for(unsigned int i=0; i < itfdisplacement[0].size(); i++) {
        double y = mesh->regioninterface(itflist[0])->vertex(i)->coord()[1];
        itfdisplacement[0][i].resize(3);
        itfdisplacement[0][i][0] = 0.0;
        itfdisplacement[0][i][1] = 0.0;
        // Computation of the vertical displacement of the i-th grid vertex
        itfdisplacement[0][i][2] = max_displ *
            ( sin(2*pi*n/N) - sin((2*pi*(n-1)/N)) ) +
            max_displ *
            ( sin(2*pi*n/N) - sin(2*pi*(n-1)/N) ) * sin(2*pi/(y_{\text{max}}-y_{\text{min}}) * (y_{\text{c}}-y_{\text{c}}));
    }
    // End of the loop over the grid vertices
    if ( n > 0 )
        r = mesh->moving_interface(itflist, itfdisplacement,1.5);  
```

Once we defined the current imposed displacements we can call the method `moving_interface`

```cpp
    if ( n > 0 )
        r = mesh->moving_interface(itflist, itfdisplacement,1.5);  
```

After calling the method, we compute, for each grid element, the radii of its inscribed and circumscribed spheres, and its minimum and maximum dihedral angles, using the methods `radii` and `angles` of the class `agr_element`. Furthermore we calculate the minimum and the average ratio $\bar{r}/R$ and the average small and large dihedral angles, $\bar{\gamma}_{\text{min},\text{avg}}$ and $\bar{\gamma}_{\text{max},\text{avg}}$, among all the grid elements.

```cpp
double minrR(0.0), avgrR(0.0);
```
double gammaminave(0.0), gammamaxave(0.0), gammaminmin(0.0), gammamaxmax(0.0);
pair<double,double> coppia;
coppia = mesh->element(0)->angles();
gammaminmin = coppia.first;
gammamaxmax = coppia.second;
gammaminave = coppia.first;
gammamaxave = coppia.second;
coppia = mesh->element(0)->rays();
minrR = coppia.second / coppia.first;
avgrR = coppia.second / coppia.first;
// Loop over the grid elements
for (unsigned int j=1; j < nE; j++) {
    coppia = mesh->element(j)->angles();
gammaminmin = min(coppia.first,gammaminmin);
gammamaxmax = max(coppia.second,gammamaxmax);
gammaminave += coppia.first;
gammamaxave += coppia.second;
coppia = mesh->element(j)->rays();
minrR = min(coppia.second / coppia.first, minrR);
avgrR += coppia.second / coppia.first;
} // End of the loop over the grid elements
} // End of the loop over time

Figures 2.3- 2.7 show the longitudinal sections of the structure at equidistant instants in time: $t = 0$, $t = \frac{T}{4}$, $t = \frac{T}{2}$, $t = \frac{3T}{4}$ and $t = T$. As we expected meshes at $t = 0$ and at $t = T$ are the same. Figure 2.8 shows the maximum and minimum average dihedral angles through time. Figure 2.9 shows the minimum and average ratio of the radii of the inscribed and circumscribed spheres through time. Figure 2.10 shows the total number of iterations performed by the method through time. All these graphs have a periodic behavior. As we can see, the maximum and minimum average dihedral angles belongs to an acceptable range of values, they are not too small nor too large. The ratio of the radii of the inscribed and circumscribed spheres is quite low but it is not a consequence of the method, but rather an intrinsic characteristic of the grid. Moreover we can observe that the average ratio of the radii remains approximately constant. Finally we note that the number of iterations performed at each time step is rather confined and periodic. According to the considerations made so far, we can state that the method `moving_interface` behaves well, and is able to guarantee nicely graded, good quality grids even after repeated cycles of fairly severe deformations.

Figure 2.3: $t = 0$. 

![Figure 2.3: $t = 0$.]
Figure 2.4: $t = T/4$.

Figure 2.5: $t = T/2$.

Figure 2.6: $t = 3T/4$.

Figure 2.7: $t = T$. 

53
Figure 2.8: Maximum and minimum average dihedral angle versus time steps.

Figure 2.9: Minimum and average ratio of the radii of the inscribed and circumscribed spheres versus time steps.

Figure 2.10: Number of total iterations performed by the method versus time steps.
2.3.2 Application examples

Test case 1

The structure of this test case is a parallelepiped with 7 regions. Figure 2.11 shows the lower part of this structure, the seventh region, which can not be seen in the figure, is located above and has a height equal to 0.5.

In this case we reduce the region consisting in the inner parallelepiped (orange region). The displacement is not periodical. The reduction is progressive and constant through time. The total movement is subdivided in 40 times steps. Each dimension of the inner parallelepiped is reduced by half. Thus this reduction is isotropic. Figure 2.12- 2.16 show the reduction movement in time.

Test case 2

The structure of this test case is a parallelepiped with two concentric cylinders inside. The are six regions. Figure 2.17(a) shows the longitudinal section, while Fig. 2.17(b) shows the cross section of the structure.

In this case we reduce along the radial direction the inner cylinder (red region), while we expand, again along the radial direction, the outer cylinder (orange region). The radius of the inner cylinder is reduced by a factor 1/3, while the radius of the outer cylinder is increased by a factor 1/4. The movements are progressive and constant through time. They are subdivided in 40 times steps. Figure 2.18- 2.22 show the displacements in time.
Figure 2.13: Time step 10.

Figure 2.14: Time step 20.

Figure 2.15: Time step 40.

Figure 2.16: Time step 0.
(a) Longitudinal section of the structure.

(b) Cross section of the structure.

Figure 2.17

Figure 2.18: Time step 0.

Figure 2.19: Time step 10.
Figure 2.20: Time step 20.

Figure 2.21: Time step 30.
Figure 2.22: Time step 40.
Bibliography


