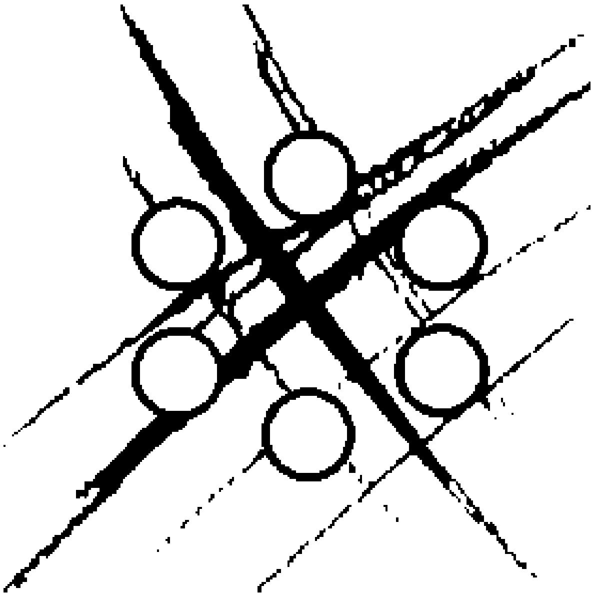
DIRECTION DE L'ÉNERGIE NUCLEAIRE

DÉPARTEMENT MODÉLISATION DE SYSTÈMES ET STRUCTURES

SERVICE FLUIDES NUMÉRIQUES, MODÉLISATION ET ÉTUDES



RAPPORT DM2S

SFME/LGLS/RT/07-001/A

MEDMEM user's guide

Vincent Bergeaud, Nadir Bouhamou

COMMISARIAT A L'ENERGIE ATOMIQUE

DIRECTION DE L'ENERGIE NUCLEAIRE
DEPARTEMENT MODELISATION DE SYSTEMES ET STRUCTURES
SERVICE FLUIDES NUMÉRIQUES, MODÉLISATION ET ÉTUDES
LABORATOIRE DE GÉNIE LOGICIEL ET SIMULATION

RAPPORT DM2S

REFERENCES : SFME/LGLS/RT/07-001/A

TITRE : MEDMEM user's guide

AUTEURS	SIGNATURES	AUTEURS	SIGNATURES
Vincent Bergeaud		Nadir Bouhamou	

RESUME : Ce document décrit les fonctionnalités de gestion de maillages offertes par la bibliothèque MED-mémoire. Il permet à un utilisateur d'exploiter les données contenues dans un maillage. Le document explique également comment créer des structures MED à partir des structures internes des codes.

MOTS CLES : Salome, maillage, intégration, structure d'échange
AFFAIRE : DSOE/informatique Projet : NEPAL EOTP : A-NPPAL-02-01
Titre de l'action : Environnement PAL

			Visa			
			Nom	N. Crouzet		L. Dada
A	05/01/2007	74	Date			
Indice	**Date**	**Nb. Pages**		**Vérificateur**	**Autre visa**	**Approbateur**

Liste des modifications

Indice	Date	Motif et description de la modification
A	05/01/2007	Document initial

Contents

1 Introduction

1.1 Rationale for Med Memory

The Med data exchange model (DEM in English) is the format used in the Salome platform for communicating data between different components. It manipulates objects that describe the meshes underlying scientific computations and the value fields lying on these meshes. This data exchange can be achieved either through files using the Med-file formalism or directly through memory with the Med Memory (`MEDMEM`) library.
The Med libraries are oganized in multiple layers:

- The MED file layer : C and Fortran API to implement mesh and field persistency.
- The MED Memory level C++ API to create and manipulate mesh and field objects in memory.
- Python API generated using SWIG which wraps the complete C++ API of the MED Memory
- CORBA API to simplify distributed computation inside SALOME (Server Side).
- MED Client classes to simplify and optimize interaction of distant objects within the local solver.

Thanks to Med Memory, any component can access a distant mesh or field object. Two codes running on different machines can thus exchange meshes and fields. These meshes and fields can easily be read/written in a Med file format, enabling access to the whole Salome suite of tools (CAD, meshing, Visualization, other components).

1.2 Outline

In this document, we describe the API of the Med Memory library (available in C++ and in Python). This document is intended for developers who are in charge of integrating existing applications in the Salome platform.
As will be seen in section 2, the API consists of very few classes:

- a general MED container,
- meshes,
- supports and derived classes,
- fields
- drivers for reading and writing in MED, GIBI and VTK files.

All these are detailed in the following sections. The C++ formalism will be used for the description in these sections. Python syntax is very similar and is given in appendix 10.

1.3 Naming conventions

The naming conventions are rather straightforward, but the user familiar with the Med-File semantics may find that there are a few noticeable differences (see the following section).

cell entity of dimension equal to the mesh dimension (1, 2 or 3).

component in a field, represents a value that is available for each element of the support (for instance : T, v_x, σ_{xy})).

connectivity (descending) connectivity table expressing connectivity of dimension d elements in terms of list of dimension $d-1$ elements.

connectivity (nodal) connectivity table expressing connectivity of dimension d elements in terms of list of nodes.

constituent entity entity having a dimension smaller than that of the mesh.

coordinates in a mesh, coordinates can be described by strings giving the names of the coordinates, the units of the coordinates, and the type of coordinates ('MED_CART', 'MED_SPHER' or 'MED_CYL').

description string of characters used to describ an object without giving any access to a query method.

dimension Med Memory discriminates the mesh dimension from the space dimension (a surface shape in $3D$ will have 2 as a mesh dimension).

driver object attached to a mesh or a field to read (resp. write) data from (resp. to) a Med-file.

edge entity of dimension 1 in a $2D$ mesh.

element elementary component of a mesh ($0D$, $1D$, $2D$ or $3D$).

entity category giving information on the dimension of elementary components of meshes : node, edge, face (only in $3D$) or cell.

face for $3D$ meshes, faces are the $2D$ entities.

family support which is composed of a set of groups, which do not intersect each other, and which gives access to those groups.

field array of integer, integer array, real or real array lying on a support (the dimension of the array of values for each element of the support is called the number of components). A field is uniquely defined by its name, its support, its iteration number and its order number. -1 is the default value of those two numbers.

group support with additional access to parent families.

iteration number information attached to a field that expresses the number of the time step in the computation (-1 is its default value).

name information attached to a mesh, support or field to name it and access to it.

node entity of dimension 0.

order number information attached to a field that expresses the number of an internal iteration inside a time step in the computation (-1 is its default value).

support list of elements of the same entity.

type category of an entity (triangle, segment, quadrangle, tetrahedron, hexahedron, etc...).

1.4 Differences with Med-File concepts

Though the MEDMEM library can recompute a descending connectivity from a nodal connectivity, MEDMEM drivers can only read MED files containing the nodal connectivities of the entities. In MEDMEM, constituent entities are stored as `MED_FACE` or `MED_EDGE`, whereas in MED File, they should be stored as `MED_MAILLE`.

The field notion in MED File and MEDMEM is quite different. In MEDMEM a field is of course defined by its name, but also by its iteration number and its order number. In MED File a field is only flagged by its name. For instance, a temperature at times *t=0.0 s, t=1.0 s, t=2.0 s* will be considered as a single field in Med File terminology, while it will be considered as three distinct fields in the Med Memory sense.

2 Med Memory API

2.1 Conventions

- In this document, one refers to the main user documentation [2] where the variable `$MED_ROOT_DIR` (resp. `$MED_SRC_DIR`) is the Med Memory directory installation (resp. sources directory).

- All numberings start at one (take care of array index !).

- When one gets a C (resp. C++) type array (resp. STL container) using a `get...` method, one should not modify the array. Access is in read only. To modify a such array (resp. STL container) use a `set...` method.

- There are many couple of methods that have similar syntaxes (one singular and one plural). The plural method returns an array and the singular one returns one particular value in this array (see method **double getCoordinate(int i)** and method **double* getCoordinates()** for example). Generally, only the plural version of the methods are documented in this report.

- Difference between local and global number in mesh element connectivity list : when one talks about an element number, one could see i^{th} quadrangle (i^{th} in quadrangles array : local numbering) or j^{th} element (j^{th} in all elements array : global numbering). These two numberings are equivalent only if one has only one geometric type.

2.2 Namespaces

Med Memory uses two namespaces : `MEDMEM` which is the general namespace where the main classes are defined and `MED_EN` which defines enums that can be used by an English-speaking programer.

2.3 Classes

At a basic usage level, the API consists in few classes which are located in the `MEDMEM` C++ namespace (consult figure 1 which gives an UML diagram view of the main Med Memory classes) :

MED the global container;

MESH the class containing 2D or 3D mesh objects;

SUPPORT the class containing mainly a list of mesh elements;

FIELD the class template containing list of values lying on a particular support.

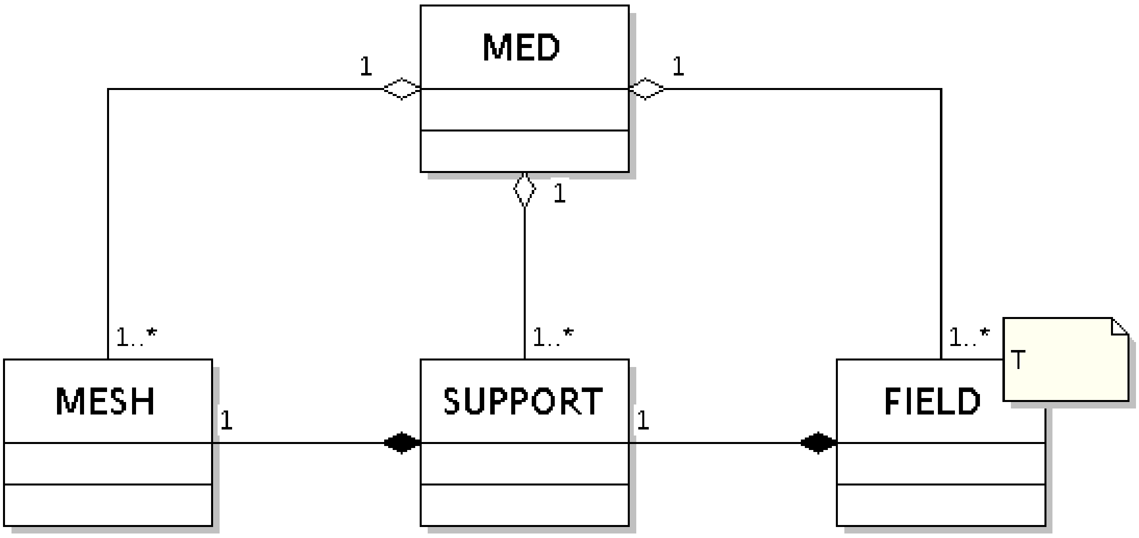


Figure 1: UML diagram of basic Med Memory API classes.

The API of those classes is quite sufficient for most of the component integrations in the Salome platform. The use of the Med Memory libraries may make easier the code coupling in the Salome framework. With these classes, it is possible to :

- read/write meshes and fields from MED-files;

- create fields containing scalar or vectorial values on list of elements of the mesh;
- communicate these fields between different components;
- read/write such fields.

Note that on the figure 1 as well as on figure 2 that the MED container controls the life cycle of all the objects it contains : its destructor will destroy all the objects it aggregates. On the other hand, the life cycle of mesh, support and field objects are independent. Destroying a support (resp. a field) will have no effect on the mesh (resp. support) which refers to it. But the user has to maintain the link : a mesh aggregates a support which aggregates a field. If the user has to delete Med Memory objects, the field has to be deleted first, then the support and finally the mesh.
A more advanced usage of the Med Memory is possible through other classes. Figure 2 gives a complete view of the Med Memory API. It includes :

GROUP a class inherited from the SUPPORT class used to create supports linked to mesh groups. It stores restricted list of elements used to set boundary conditions, initial values.

FAMILY which is used to manipulate a certain kind of support which does not intersect each other;

MESHING which builds meshes from scratch, it can be used to transform meshes from a specific format to the MED format or to integrate a mesher within Salome platform (note that class does not add element or node to a mesh);

GRID which enables the user to manipulate specific functions for structured grid.

2.4 Enums

A few enums are defined in the `MED_EN` namespace :

- an enum which describes the way node coordinates or field values are stored,
  - `MED_FULL_INTERLACE` for arrays such that $x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n$;
  - `MED_NO_INTERLACE` for arrays such that $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_n$;
  - `MED_UNDEFINED_INTERLACE`, the undefined interlacing mode.
- an enum which describes the type of connectivity
  - `MED_NODAL` for nodal connectivity;
  - `MED_DESCENDING` for descending connectivity.

  The user has to be aware of the fact that the Med Memory considers only meshes defined by their nodal connectivity. Nevertheless, the user may, after loading a file in memory, ask to the mesh object to calculate the descending connectivity.

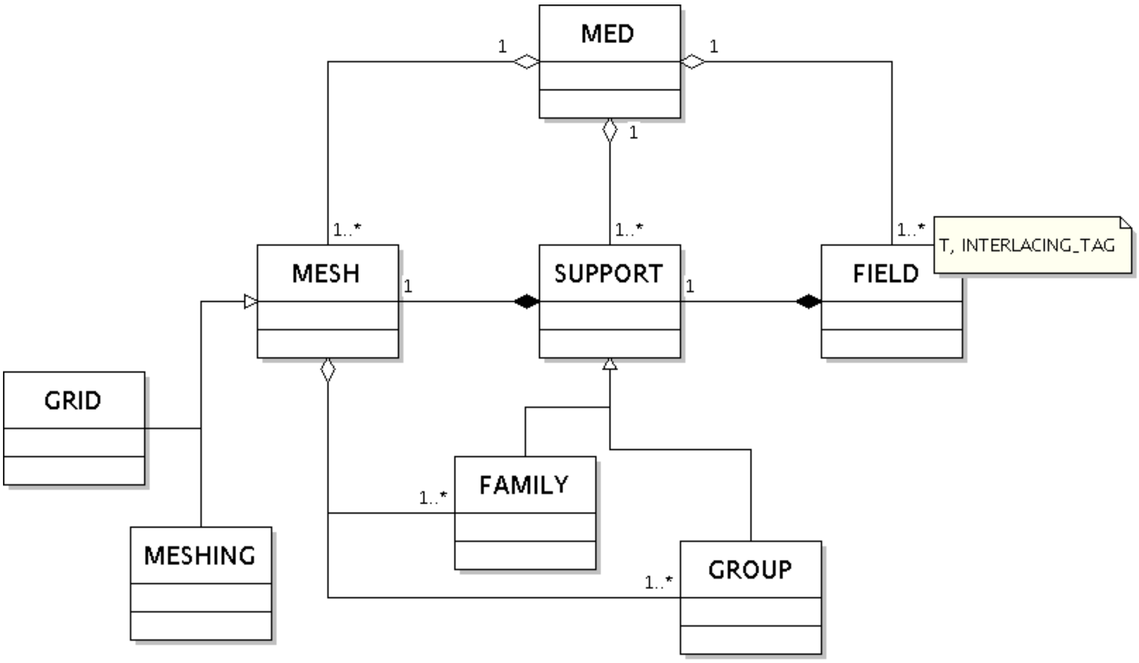


Figure 2: UML diagram of Med Memory API classes.

- an enum which contains the different mesh entities, `medEntityMesh`, the entries of which being :
  - `MED_CELL`
  - `MED_FACE`
  - `MED_EDGE`
  - `MED_NODE`
  - `MED_ALL_ENTITIES`

  In 3 (resp. 2) D, the user has to be aware of the fact that only mesh entities `MED_CELL` and `MED_FACE` (resp. `MED_EDGE`) are considered. In 1 D, of course only mesh entities `MED_CELL` are considered. Using our naming convention (consult 1.3), in 1 D mesh only **node** and **cell** are considered. In 2 D mesh, only **node**, **cell** and **edge** are considered. Finally in 3 D mesh only **node**, **cell** and **face** are considered.

- The `medGeometryElement` enum which defines geometric types. The available types are linear and quadratic elements (consult [2]). The entries of this enum are quite self-explanatory :
  - `MED_NONE`
  - `MED_POINT1`
  - `MED_SEG2`
  - `MED_SEG3`

- `MED_TRIA3`
- `MED_QUAD4`
- `MED_TRIA6`
- `MED_QUAD8`
- `MED_TETRA4`
- `MED_PYRA5`
- `MED_PENTA6`
- `MED_HEXA8`
- `MED_TETRA10`
- `MED_PYRA13`
- `MED_PENTA15`
- `MED_HEXA20`
- `MED_POLYGON`
- `MED_POLYHEDRA`
- `MED_ALL_ELEMENTS`

The connectivity of all these elements is defined in document [4].

3 MESH

3.1 General information

The MESH class is dedicated to the handling of unstructured meshes. Two classes derive from it : MESHING supplies functions for creating meshes from scratch (c.f. 8), while GRID gives specific constructors for creating structured meshes.

3.2 Content of the connectivity array

Underlying the unstructured meshes is the notion of connectivity. This section only covers meshes made out of standard elements, the `MED_POLYGON` and `MED_POLYHEDRA` case being detailed in section 9

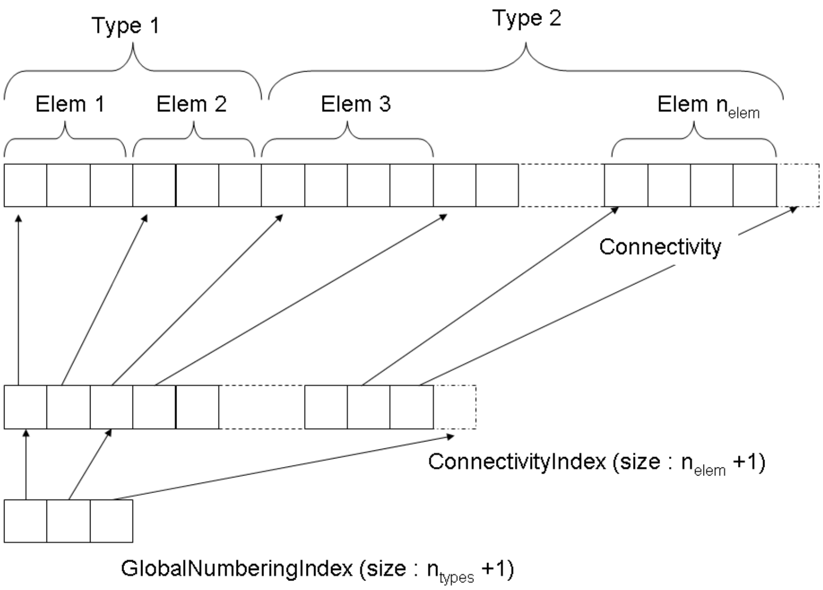

Type 1
Type 2
Elem 1
Elem 2
Elem 3
Elem n_{elem}
Connectivity
ConnectivityIndex (size : n_{elem} +1)
GlobalNumberingIndex (size : n_{types} +1)

Figure 3: Nodal connectivity storage scheme

In MEDMEM, an unstructured mesh nodal connectivity is defined with these arrays (if the mesh has no MED_POLYGON and MED_POLYHEDRA element) :

- the type array, which contains the number of cells for each present type
- the nodal connectivity array containing the connectivity of each cell, all cells being sorted by type,

- the connectivity index array, which indicates the beginning of each cell in the connectivity array,

The cell types are ordered by their number of nodes.
As an example, let us consider a mesh made out of a linear triangle, two linear quadrangles and a quadratic triangle (c.f. figure 4).

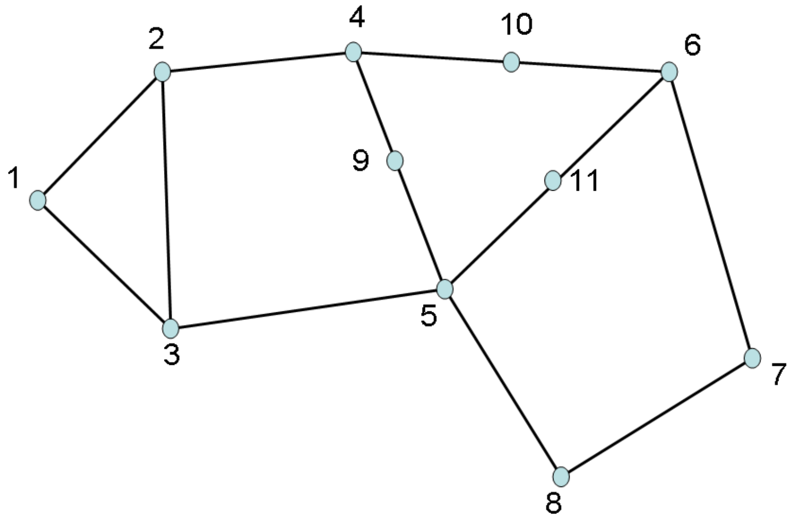


Figure 4: Example for mesh connectivity

The number of types is : *3*

The type array writes : *{ MED_TRIA3, MED_QUAD4, MED_TRIA6}*

The global numbering index is : *{1,2,4,5}* . Its dimension is $n_{types} + 1$ so that elements of type $type[i]$ are stored between element $index[i]$ and $index[i+1]$ ($index[i] \leq j < index[i+1]$).

The connectivity array writes : *{ 1, 2, 3, 2, 4, 5, 3, 5, 6, 7, 8, 4, 6, 5, 10, 11, 9}*

The connectivity index array writes : *{ 1, 4, 8, 12, 18}*
Its dimension is $n_{cell} + 1$, in order to be able to write that nodes of element i are located in the connectivity array between $index[i]$ and $index[i+1]$ ($index[i] \leq j < index[i+1]$).

Warning : As MEDMEM respects MED numbering which starts Fortran-style at 1, reading these information to set C structures requires careful handling of index offsets.

3.3 Constructors

Functions

- **MEDMEM::MESH::MESH** (MESH &m)
- **MEDMEM::MESH::MESH** (driverTypes driverType, const string &fileName="", const string &meshName="") throw (MEDEXCEPTION)

3.3.1 Detailed Description

The MESH class provides only two constructors : a copy constructor and a constructor enabling creation from file reading. The creation of a user-defined mesh implies the use of the MESHING class.

3.3.2 Function Documentation

MESH (MESH & *m*) `[inherited]`
Copy constructor

MESH (driverTypes *driverType*, const string & *fileName* = "", const string & *driverName* = "") throw (MEDEXCEPTION) `[inherited]`
Create a MESH object using a MESH driver of type *driverType* (MED_DRIVER, ...) associated with file fileName. The meshname *driverName* must exist in the file.

Parameters:

driverType file type (MED_DRIVER, VTK_DRIVER, GIBI_DRIVER or PORFLOW_DRIVER)

fileName name and path of the file

driverName mesh name

3.4 General information

Functions

- string **MEDMEM::MESH::getName** () const
- string **MEDMEM::MESH::getDescription** () const
- int **MEDMEM::MESH::getSpaceDimension** () const
- int **MEDMEM::MESH::getMeshDimension** () const

3.4.1 Detailed Description

These methods are related to the retrieval of general information about the mesh.

3.4.2 Function Documentation

string getName () const `[inline, inherited]`
Gets the MESH name.

string getDescription () const `[inline, inherited]`
Gets the MESH description. The string returned contains a short description of the mesh, which is stored for information purposes only.

int getSpaceDimension () const `[inline, inherited]`
Gets the dimension of the space in which the mesh is described (2 for planar meshes, 3 for volumes and 3D surfaces) .

int getMeshDimension () const `[inline, inherited]`
Gets the dimension of the mesh (2 for 2D- and 3D-surfaces, 3 for volumes).

3.4.3 Example

Here is a small C++ example program, the Python version of which may be found in
10.1.

```
// Copyright (C) 2005  OPEN CASCADE, EADS/CCR, LIP6, CEA/DEN,
// CEDRAT, EDF R&D, LEG, PRINCIPIA R&D, BUREAU VERITAS
//
using namespace std;
#include "MEDMEM_Mesh.hxx"

using namespace MEDMEM ;

int main (int argc, char ** argv) {

  const string MedFile = "pointe.med" ;
  const string MeshName = "maa1" ;

  // create a MESH object by reading it on file :
  MESH myMesh(MED_DRIVER,MedFile,MeshName) ;

  string Name = myMesh.getName() ;
  if (Name != MeshName) {
    cout << "Error when reading mesh name : We ask for mesh #"
 << MeshName <<"# and we get mesh #"<< Name <<"#"<< endl << endl ;
    return -1;
  }

  cout << "Mesh name : " << Name  << endl << endl ;

  int SpaceDimension = myMesh.getSpaceDimension() ;
  int MeshDimension = myMesh.getMeshDimension() ;

  cout << "Space Dimension : " << SpaceDimension << endl << endl ;
  cout << "Mesh Dimension : " << MeshDimension << endl << endl ;

  return 0 ;
}
```

3.5 Coordinates information

Functions

- int **MEDMEM::MESH::getNumberOfNodes** () const
- string **MEDMEM::MESH::getCoordinatesSystem** () const
- virtual const double ∗ **MEDMEM::MESH::getCoordinates** (MED_EN::medModeSwitch Mode) const
- virtual const double **MEDMEM::MESH::getCoordinate** (int Number, int Axis) const
- const string ∗ **MEDMEM::MESH::getCoordinatesNames** () const
- const string ∗ **MEDMEM::MESH::getCoordinatesUnits** () const

3.5.1 Detailed Description

These methods are related to the extraction of information about the mesh nodes coordinates.

3.5.2 Function Documentation

int getNumberOfNodes () const `[inline, inherited]`
Gets the number of nodes used in the mesh.

string getCoordinatesSystem () const `[inline, inherited]`
Retrieves the system in which coordinates are given (MED_CART,MED_CYL,MED_SPHER).

const double ∗ getCoordinates (MED_EN::medModeSwitch *Mode*) const `[inline, virtual, inherited]`
Gets the whole coordinates array in a given interlacing mode. The interlacing mode are :

- MED_NO_INTERLACE : X1 X2 Y1 Y2 Z1 Z2
- MED_FULL_INTERLACE : X1 Y1 Z1 X2 Y2 Z2

const double getCoordinate (int *number*, int *axis*) const `[inline, virtual, inherited]`
Gets the coordinate number *number* on axis *axis*.

const string ∗ getCoordinatesNames () const `[inline, inherited]`
Gets a pointer to the coordinate names array.

const string ∗ getCoordinatesUnits () const `[inline, inherited]`
Gets a pointer to the coordinate units array.

3.5.3 Example

Here is a small C++ example program for which the Python version may be found in 10.2.

```
// Copyright (C) 2005  OPEN CASCADE, EADS/CCR, LIP6, CEA/DEN,
// CEDRAT, EDF R&D, LEG, PRINCIPIA R&D, BUREAU VERITAS
//
#include "MEDMEM_Mesh.hxx"

using namespace MEDMEM ;
using namespace MED_EN ;

int main (int argc, char ** argv) {

  const string MedFile = "pointe.med" ;
  const string MeshName = "maa1" ;
  MESH myMesh(MED_DRIVER,MedFile,MeshName) ;

  cout << "Mesh name : " << myMesh.getName()  << endl << endl ;

  int SpaceDimension = myMesh.getSpaceDimension() ;
  int NumberOfNodes = myMesh.getNumberOfNodes() ;
  cout << "Space dimension  : " << SpaceDimension << endl << endl ;
  cout << "Number of nodes  : " << NumberOfNodes  << endl << endl ;

  cout << "Show Nodes Coordinates : " << endl ;
  // coordinates names :
  cout << "Name :" << endl ;
  const string * CoordinatesNames = myMesh.getCoordinatesNames() ;
  for(int i=0; i<SpaceDimension ; i++) {
    cout << " - " << CoordinatesNames[i] << endl ;
  }
  // coordinates unit :
  cout << "Unit :" << endl ;
  const string * CoordinatesUnits = myMesh.getCoordinatesUnits() ;
  for(int i=0; i<SpaceDimension ; i++) {
    cout << " - " << CoordinatesUnits[i] << endl ;
  }
  // coordinates value
  const double * Coordinates =
    myMesh.getCoordinates(MED_FULL_INTERLACE) ;
  for(int i=0; i<NumberOfNodes ; i++) {
    cout << "Nodes " << i+1 << " : " ;
    for (int j=0; j<SpaceDimension ; j++)
      cout << Coordinates[i*SpaceDimension+j] << " " ;
    cout << endl ;
  }

  return 0 ;
}
```

3.6 Connectivity information

Functions

- virtual int **MEDMEM::MESH::getNumberOfTypes** (MED_EN::medEntityMesh Entity) const
- virtual const MED_EN::medGeometryElement * **MEDMEM::MESH::getTypes** (MED_EN::medEntityMesh Entity) const
- virtual const int * **MEDMEM::MESH::getGlobalNumberingIndex** (MED_EN::medEntityMesh Entity) const
- virtual int **MEDMEM::MESH::getNumberOfElements** (MED_EN::medEntityMesh Entity, MED_EN::medGeometryElement Type) const
- virtual MED_EN::medGeometryElement **MEDMEM::MESH::getElementType** (MED_EN::medEntityMesh Entity, int Number) const
- virtual const int * **MEDMEM::MESH::getConnectivity** (MED_EN::medModeSwitch Mode, MED_EN::medConnectivity ConnectivityType, MED_EN::medEntityMesh Entity, MED_EN::medGeometryElement Type) const
- virtual const int * **MEDMEM::MESH::getConnectivityIndex** (MED_EN::medConnectivity ConnectivityType, MED_EN::medEntityMesh Entity) const

3.6.1 Detailed Description

These methods are related to the extraction of connectivity information from the mesh.

3.6.2 Function Documentation

int getNumberOfTypes (MED_EN::medEntityMesh *entity*) const `[inline, virtual, inherited]`

Gets the number of different geometric types for a given entity type.

For example getNumberOfTypes(MED_CELL) would return 3 if the MESH have some MED_TETRA4, MED_PYRA5 and MED_HEXA8 in it. If entity is not defined, returns 0. If there is no connectivity, returns an exception.

Parameters:

entity entity type (MED_CELL, MED_FACE, MED_EDGE, MED_NODE, MED_ALL_ENTITIES)

const MED_EN::medGeometryElement $*$ getTypes (MED_EN::medEntityMesh *entity*) const `[inline, virtual, inherited]`

Gets the list of geometric types used by a given entity. If entity is not defined, it returns an exception.

Parameters:

entity Entity type must be MED_CELL, MED_FACE, MED_EDGE or MED_ALL_ENTITIES. Passing MED_NODE as an entity type will throw an exception.

const int $*$ getGlobalNumberingIndex (MED_EN::medEntityMesh *entity*) const `[inline, virtual, inherited]`

Returns an array of size NumberOfTypes+1 which contains, for each geometric type of the given entity, the first global element number of this type.

For exemple, if we have a mesh with 5 triangles and 4 quadrangle :

- size of GlobalNumberingIndex is 3
- GlobalNumberingIndex[0]=1 (the first type)
- GlobalNumberingIndex[1]=6 (the second type)
- GlobalNumberingIndex[2]=10

int getNumberOfElements (MED_EN::medEntityMesh *entity*, MED_EN::medGeometryElement *Type*) const `[inline, virtual, inherited]`

Returns the number of elements of given geometric type of given entity. Returns 0 if query is not defined.

Example :

- getNumberOfElements(MED_NODE,MED_NONE) : number of nodes
- getNumberOfElements(MED_NODE,MED_TRIA3) : returns 0 (not defined)
- getNumberOfElements(MED_FACE,MED_TRIA3) : returns number of triangle elements defined in face entity (0 if not defined)
- getNumberOfElements(MED_CELL,MED_ALL_ELEMENTS) : returns total number of elements defined in cell entity

MED_EN::medGeometryElement getElementType (MED_EN::medEntityMesh *Entity*, int *Number*) const `[inline, virtual, inherited]`

Returns the geometric type of global element number *Number* of entity *Entity*.

Throw an exception if *Entity* is not defined or if *Numberis* wrong.

const int ∗ getConnectivity (MED_EN::medModeSwitch *Mode*, MED_EN::medConnectivity *ConnectivityType*, MED_EN::medEntityMesh *entity*, MED_EN::medGeometryElement *Type*) const `[inline, virtual, inherited]`

Returns the required connectivity in mode *Mode* for the geometric type *Type* of the entity type *entity*. *ConnectivityType* specifies descending or nodal connectivity.

To get connectivity for all geometric type, use *Mode=MED_FULL_INTERLACE* and *Type=MED_ALL_ELEMENTS*. You must also get the corresponding index array.

const int ∗ getConnectivityIndex (MED_EN::medConnectivity *ConnectivityType*, MED_EN::medEntityMesh *entity*) const `[inline, virtual, inherited]`

Returns the required index array for a connectivity received in MED_FULL_INTERLACE mode and MED_ALL_ELEMENTS type.

This array allows to find connectivity of each element.

Example : Connectivity of i-th element (1<=i<=NumberOfElement) begins at index ConnectivityIndex[i-1] and ends at index ConnectivityIndex[i]-1 in Connectivity array (Connectivity[ConnectivityIndex[i-1]-1] is the first node of the element)

3.6.3 Example

This example shows the use of connectivity retrieval methods on a mesh which corresponds to the four-element mesh given in figure 4. Note the use of connectivity and connnectivity index tables, and the offsets used to convert Fortran-style numbering to C arrays.

```
#include "MEDMEM_Mesh.hxx"
using namespace MEDMEM;
using namespace MED_EN;

int main() {
  const string MedFile="example.med";
  const string MeshName="meshing";
  MESH my_mesh(MED_DRIVER, MedFile, MeshName);

// Type retrieval
  int number_of_types=my_mesh.getNumberOfTypes(MED_CELL);
  cout << "Number of types :"<< number_of_types<<endl;

  const medGeometryElement * Types = my_mesh.getTypes(MED_CELL);
    const int * connectivity_index=
      my_mesh.getConnectivityIndex( MED_NODAL,
     MED_CELL
     );
   const int * connectivity =
      my_mesh.getConnectivity(MED_FULL_INTERLACE,
     MED_NODAL,
     MED_CELL,
     MED_ALL_ELEMENTS);

   const int* global_index=my_mesh.getGlobalNumberingIndex(MED_CELL);

  for (int i=0; i<number_of_types; i++) {
    cout <<"Type #"<<i<<endl;
    medGeometryElement myType = Types[i] ;
    int NumberOfElements = my_mesh.getNumberOfElements(MED_CELL,myType);

// Connectivity retrieval

    for (int j= global_index[i]; j<global_index[i+1]; j++){
      cout << "Element "<< j <<" : " ;
      //global_index starts at one, so connectivity_index[j-1]
      //must be used in order to retrieve the correct node number

      for (int k=connectivity_index[j-1]; k<connectivity_index[j]; k++)
        {
      //connectivity_index starts at one, so connectivity[k-1]
      //must be used in order to retrieve the correct node number
  cout << connectivity[k-1]<<" ";
         }
      cout << endl ;
    }
  }
}
```

The output of this program reads :

```
Number of types : 3
Type #0
Element 1 : 1 2 3
Type #1
Element 1 : 2 4 5 3
Element 2 : 5 6 7 8
Type #2
Element 1 : 4 6 5 10 11 9
```

A more complete example involving descending connectivities can be found in `MESHconnectivities.cxx` and `MESHconnectivities.py`.

3.7 Group management

Functions

- virtual const vector< GROUP ∗ > **MEDMEM::MESH::getGroups** (MED_EN::medEntityMesh Entity) const

3.7.1 Detailed Description

These methods describe how to access the groups that are defined on the mesh. This user's guide does not fully describe the Family notion that can also be used with Medmem for fine tuning of the memory usage. More information can be found on this topic in the Developer's guide. Groups are defined on a specific entity (MED_-CELL, MED_FACE, MED_EDGE or MED_NODE). Therefore, the access method defined in this subsection is passed an entity type as an argument.

3.7.2 Function Documentation

const vector< GROUP ∗ > getGroups (MED_EN::medEntityMesh *entity*) const `[inline, virtual, inherited]`
Returns the groups of type *entity* present in the mesh as a vector of pointers. The GROUP class inheriting from the SUPPORT class, the methods that can be used on these groups are explained in the related section.

3.8 File I/O

Functions

- int **MEDMEM::MESH::addDriver** (driverTypes driverType, const string &fileName="Default File Name.med", const string &driverName="Default Mesh Name", MED_EN::med_mode_acces access=MED_EN::MED_REMP)
- void **MEDMEM::MESH::write** (int index=0, const string &driverName="")

3.8.1 Detailed Description

File reading should be done through a MESH constructor. For file writing, two methods have to be used : one for driver initialization, the second one triggering the writing of the file. The MED_DRIVER type is used to specify the format of the input/output file. It can be one of the following values :

- MED_DRIVER,
- GIBI_DRIVER (read only),
- VTK_DRIVER (write only),
- PORFLOW_DRIVER (read only).

3.8.2 Function Documentation

int addDriver (driverTypes *driverType*, const string & *fileName* = `"Default File Name.med"`**, const string & *driverName* =** `"Default Mesh Name"`**, MED_EN::med_mode_acces *access* =** `MED_EN::MED_REMP`**)** `[inherited]`
Add a MESH driver of type *driverType* (MED_DRIVER, ...) associated with file *fileName*. The meshname used in the file is *driverName*. The mode access *med_mode_access* gives by default read and write permissions. The return value of *addDriver* is an integer handle.

void write (int *index* = `0`**, const string & *driverName* =** `""`**)** `[inline, inherited]`
Writes all the content of the MESH using driver referenced by the integer handle returned by a *addDriver* call.
Example :

```
//...
// Attaching the driver to file "output.med", meshname "Mesh"
int driver_handle = mesh.addDriver(MED_DRIVER, "output.med", "Mesh");
// Writing the content of mesh to the file
mesh.write(driver_handle);
```

3.9 Advanced features

Functions

- virtual SUPPORT ∗ **MEDMEM::MESH::getBoundaryElements** (MED_EN::medEntityMesh Entity) throw (MEDEXCEPTION)
- virtual FIELD< double > ∗ **MEDMEM::MESH::getVolume** (const SUPPORT ∗Support) const throw (MEDEXCEPTION)
- virtual FIELD< double > ∗ **MEDMEM::MESH::getArea** (const SUPPORT ∗Support) const throw (MEDEXCEPTION)
- virtual FIELD< double > ∗ **MEDMEM::MESH::getLength** (const SUPPORT ∗Support) const throw (MEDEXCEPTION)
- virtual FIELD< double > ∗ **MEDMEM::MESH::getNormal** (const SUPPORT ∗Support) const throw (MEDEXCEPTION)
- virtual FIELD< double > ∗ **MEDMEM::MESH::getBarycenter** (const SUPPORT ∗Support) const throw (MEDEXCEPTION)

3.9.1 Detailed Description

These functions provide access to high-level manipulation of the meshes, giving information about the cells or extracting supports from the mesh.

3.9.2 Function Documentation

SUPPORT ∗ getBoundaryElements (MED_EN::medEntityMesh *Entity*) throw (MEDEXCEPTION) `[virtual, inherited]`

Returns a support which references all elements on the boundary of the mesh. For instance, one would obtain MED_FACE support in 3D and MED_EDGE support in 2D.

FIELD< double, FullInterlace > ∗ getVolume (const SUPPORT ∗ *Support*) const throw (MEDEXCEPTION) `[virtual, inherited]`

Calculates the volume of all the elements contained in *Support*. This method returns a FIELD structure based on this support. It only works on MED_CELL for 3D meshes.

FIELD< double, FullInterlace > ∗ getArea (const SUPPORT ∗ *Support*) const throw (MEDEXCEPTION) `[virtual, inherited]`

Calculates the area of all the elements contained in *Support*. This method returns a FIELD structure based on this support. It only works on MED_CELL for 2D meshes or MED_FACE for 3D meshes.

FIELD< double, FullInterlace > ∗ getLength (const SUPPORT ∗ *Support*) const throw (MEDEXCEPTION) `[virtual, inherited]`

Calculates the length of all the elements contained in *Support*. This method returns a FIELD structure based on this support. It only works on MED_EDGE supports.

FIELD< double, FullInterlace > ∗ getNormal (const SUPPORT ∗ *Support*) const throw (MEDEXCEPTION) `[virtual, inherited]`

Calculates the normal for all elements contained in SUPPORT *Support*. The method is only functional for 2D supports for 3D meshes and 1D supports for 2D meshes. It returns a FIELD for which the number of components is equal to the dimension of the mesh and which represents coordinates of the vector normal to the element.

In 3D, the normal vector is computed by taking three nodes in the face. Therefore, it will be coherent for planar faces only.

The direction of the vector is undetermined.

FIELD< double, FullInterlace > ∗ getBarycenter (const SUPPORT ∗ *Support*) const throw (MEDEXCEPTION) `[virtual, inherited]`

Returns the "barycenter" for each element in the support. The barycenter positions are returned as a field with a number of components equal to the mesh dimension. The barycenter is computed by an average of the node positions of the element, putting equal weight on each node.

4 GRID

4.1 General Information

The GRID class represents structured meshes in the MEDMEM library. As the GRID class inherits from MESH, all of the functionalities that were described in the previous section apply for structured mesh GRID objects. In particular, reading and writing from files, general information access are similar. However, because of the particular nature of structured meshes, there exist methods to access information related to the axes of the grid.

The numbering of the cells and nodes in a grid starts at one and the inner loop is that of the first axis, the outer loop being the one of the last axis (c.f. figure 5).

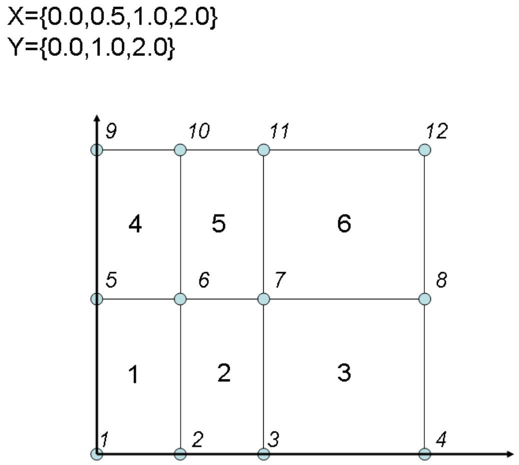


Figure 5: Example for structured mesh connectivity. The numbering is automatically defined from the two input verctors X and Y.

4.2 Constructors

Functions

- **MEDMEM::GRID::GRID** (const std::vector< std::vector< double > > &xyz_array, const std::vector< std::string > &coord_name, const std::vector< std::string > &coord_unit, const MED_EN::med_grid_type type=MED_EN::MED_CARTESIAN)

4.2.1 Detailed Description

These methods constitute the different constructors for the grid objects.

4.2.2 Function Documentation

GRID (const std::vector< std::vector< double > > & *xyz_array*, const std::vector< std::string > & *coord_name*, const std::vector< std::string > & *coord_unit*, const MED_EN::med_grid_type *type* = `MED_EN::MED_CARTESIAN`**)** `[inherited]`

Constructor specifying the axes of the grid.

This constructor describes the grid by specifying the location of the nodes on each of the axis. The dimension of the grid is implicitly defined by the size of vector *xyz_array*.

Parameters:

xyz_array specifies the node coordinates for each direction

coord_name names of the different coordinates

med_grid_type grid type (MED_POLAR, MED_CARTESIAN)

4.3 Information about axes

Functions

- int **MEDMEM::GRID::getArrayLength** (const int Axis) const throw (MEDEXCEPTION)
- const double **MEDMEM::GRID::getArrayValue** (const int Axis, const int i) const throw (MEDEXCEPTION)

4.3.1 Detailed Description

This group of methods retrieves information about the axes of the grid.

4.3.2 Function Documentation

int getArrayLength (const int *Axis*) const throw (MEDEXCEPTION) `[inherited]`

Returns the number of nodes on axis number *Axis* (axis numbering starts at 1).

const double getArrayValue (const int *Axis*, const int *i*) const throw (MEDEXCEPTION) `[inherited]`

Returns the value of node coordinate *i* on axis *Axis*.

4.4 Utility methods for defining element positions in the grid

Position to number conversion methods

getXXXNumber methods enable the user to convert an (i, j, k) position into a global number in the array. Axis [1,2,3] means one of directions: along *i*, *j* or *k*. For cell constituents (FACE or EDGE), Axis selects one of those having same (i, j, k) :

- a FACE which is normal to direction along given *Axis*;
- an EDGE going along given *Axis*.

Exception for *Axis* out of range. For 2D grids, *k* is a dummy argument.

- int **MEDMEM::GRID::getEdgeNumber** (const int Axis, const int i, const int j=0, const int k=0) const throw (MEDEXCEPTION)
- int **MEDMEM::GRID::getFaceNumber** (const int Axis, const int i, const int j=0, const int k=0) const throw (MEDEXCEPTION)

Number to position conversion methods

getXXXPosition functions enable the user to convert a number into a (i, j, k) position. Axis [1,2,3] means one of directions: along i, j or k For Cell contituents (FACE or EDGE), Axis selects one of those having same (i,j,k):

- a FACE which is normal to direction along given Axis;
- an EDGE going along given Axis.

Exception for Number out of range.

- void **MEDMEM::GRID::getNodePosition** (const int Number, int &i, int &j, int &k) const throw (MEDEXCEPTION)
- void **MEDMEM::GRID::getCellPosition** (const int Number, int &i, int &j, int &k) const throw (MEDEXCEPTION)
- void **MEDMEM::GRID::getEdgePosition** (const int Number, int &Axis, int &i, int &j, int &k) const throw (MEDEXCEPTION)
- void **MEDMEM::GRID::getFacePosition** (const int Number, int &Axis, int &i, int &j, int &k) const throw (MEDEXCEPTION)

4.4.1 Detailed Description

These methods enable the user to convert a position on the grid to a global element number

4.4.2 Function Documentation

int getEdgeNumber (const int *Axis*, const int *i*, const int *j* = `0`**, const int *k* =** `0`**) const throw (MEDEXCEPTION)** `[inherited]`

Edge position to number conversion method

int getFaceNumber (const int *Axis*, const int *i*, const int *j* = `0`**, const int *k* =** `0`**) const throw (MEDEXCEPTION)** `[inherited]`

Face position to number conversion method

void getNodePosition (const int *Number*, int & *i*, int & *j*, int & *k*) const throw (MEDEXCEPTION) `[inherited]`

Node number to position conversion method

void getCellPosition (const int *Number*, int & *i*, int & *j*, int & *k*) const throw (MEDEXCEPTION) `[inherited]`

Cell number to position conversion method

void getEdgePosition (const int *Number*, int & *Axis*, int & *i*, int & *j*, int & *k*) const throw (MEDEXCEPTION) `[inherited]`

Edge number to poistion conversion method

void getFacePosition (const int *Number*, int & *Axis*, int & *i*, int & *j*, int & *k*) const throw (MEDEXCEPTION) `[inherited]`

Face number to position convertion method

5 SUPPORT

5.1 General information

The SUPPORT class is the class representing subregions of the mesh in the MEDMEM library. A SUPPORT object groups together a set of elements that have similar entity types.

5.2 Constructors

Functions

- **MEDMEM::SUPPORT::SUPPORT** (MESH ∗Mesh, string Name="", MED_EN::medEntityMesh Entity=MED_EN::MED_CELL)
- **MEDMEM::SUPPORT::SUPPORT** (const SUPPORT &m)

5.2.1 Function Documentation

SUPPORT (MESH ∗ *Mesh*, string *Name* = "", MED_EN::medEntityMesh *Entity* = `MED_EN::MED_CELL`**)** `[inherited]`
Constructor of a support lying on mesh *Mesh*. By default, the support lies on all elements of type *Entity*. Partial support can be described using *setpartial* method.

Parameters:

Mesh Pointer to the mesh on which the support lies

Name Support name (should not exceed MED_TAILLE_NOM as defined in Med - i.e. 32 characters)

Entity Entity type of the support (MED_CELL,MED_FACE,MED_EDGE, MED_NODE)

SUPPORT (const SUPPORT & *m*) `[inherited]`
Copy constructor.

5.3 Creation methods

Functions

- void **MEDMEM::SUPPORT::setpartial** (string Description, int NumberOfGeometricType, int TotalNumberOfEntity, MED_EN::medGeometryElement ∗GeometricType, int ∗NumberOfEntity, int ∗NumberValue)
- void **MEDMEM::SUPPORT::setAll** (bool All)

5.3.1 Detailed Description

The creation of a support requires a number of information which is supplied to the MedMem library with the following methods. When the support is defined on all elements, the creation method is very simple, for the element list is implicitly defined.

5.3.2 Function Documentation

void setpartial (string *Description*, int *NumberOfGeometricType*, int *TotalNumberOfElements*, MED_EN::medGeometryElement ∗ *GeometricType*, int ∗ *NumberOfElements*, int ∗ *NumberValue*) `[inherited]`

This function allows the user to set a support not on all entities Entity, it should be used after an initialisation with the constructor SUPPORT(MESH∗ Mesh, string Name="", medEntityMesh Entity=MED_CELL) and after the call to the function setAll(false). It allocates and initialises all the attributs of the class SUPPORT.

Parameters:

- ***Description*** string describing the support for information purposes (should not exceed MED_TAILLE_DESC length - i.e. 200 characters)
- ***NumberOfGeometricType*** number of geometric types contained in the support
- ***TotalNumberOfElements*** number of elements in the support
- ***GeometricType*** array describing the geometric types (must be consistent with the entity that was passed as an argument to the support constructor)
- ***NumberOfElements*** array describing the number of elements for each type
- ***NumberValue*** array of IDs of the elements that constitute the group.

The following example refers to the mesh given in the mesh connectivity example. It creates a group containing the two cells on the right (the quadratic triangle and the quadrangle on the right).

```
// creating SUPPORT on cells with one value per cell
SUPPORT right_group(mesh, MED_CELL, 1);

string description = "right group";
int number_of_types=2;
int number_of_elements=2;
medGeometryElement geom_types[2]={MED_QUAD4, MED_TRIA6};
int number_of_elem_per_type[2]={1,1};
int number_value[2]={3,4};

//defining the region of the support
right_group.setpartial(description, number_of_types,
                       number_of_elements, geom_types,
                       number_of_elem_per_type, number_value);
```

When MED_POLYGON or MED_POLYHEDRON elements are included in the support, their global number should be given. For instance, on a mesh having ten MED_TRIA3 and five MED_POLYGON, the number of the first polygonal element is 11.

void setAll (bool *All*) `[inline, inherited]`

Creates a support on all elements of the type specified in the constructor.

Even if _isonAllElts is true, geometric types defining the FIELD's SUPPORT must be read from the SUPPORT not from the associated MESH (the geometric types defining the FIELD's SUPPORT may be a subset of the geometric types defined in the MESH even if for each SUPPORT geometric type all MESH entities are used).

5.4 Query methods

Functions

- int **MEDMEM::SUPPORT::getNumberOfElements** (MED_EN::medGeometryElement GeometricType) const throw (MEDEXCEPTION)
- const int ∗ **MEDMEM::SUPPORT::getNumberOfElements** () const throw (MEDEXCEPTION)
- virtual const int ∗ **MEDMEM::SUPPORT::getNumber** (MED_EN::medGeometryElement GeometricType) const throw (MEDEXCEPTION)
- virtual const int ∗ **MEDMEM::SUPPORT::getNumberIndex** () const throw (MEDEXCEPTION)

5.4.1 Function Documentation

int getNumberOfElements (MED_EN::medGeometryElement *GeometricType*) const throw (MEDEXCEPTION) `[inline, inherited]`

This method returns the number of all elements of the type GeometricType.

If isOnAllElements is false, it returns the number of elements in the support otherwise it returns number of elements in the mesh.

Example : number of MED_TRIA3 or MED_ALL_ELEMENTS elements in support.

Note : If SUPPORT is defined on MED_NODE, use MED_ALL_ELEMENTS as medGeometryElement GeometricType and it will return the number of nodes in the support (or in the mesh).

const int ∗ getNumberOfElements () const throw (MEDEXCEPTION) `[inline, inherited]`

Returns the total number of elements in the support.

const int ∗ getNumber (MED_EN::medGeometryElement *GeometricType*) const throw (MEDEXCEPTION) `[inline, virtual, inherited]`

If isOnAllElements is false, returns an array which contains all number of given medGeometryElement.

Numbering is global, ie numbers are bounded by 1 and MESH::getNumberOfElement(entity,MED_ALL_ELEMENTS) and not by 1 and MESH::getNumberOfElement(entity,geomElement).

Note : If SUPPORT is defined on MED_NODE, use MED_NONE medGeometryElement type.

const int ∗ getNumberIndex () const throw (MEDEXCEPTION) `[inline, virtual, inherited]`

If isOnAllElements is false, returns index of element number. Use it with getNumber(MED_ALL_-ELEMENTS).

5.5 Advanced methods

Functions

- void **MEDMEM::SUPPORT::blending** (SUPPORT ∗mySupport) throw (MEDEXCEPTION)
- void **MEDMEM::SUPPORT::getBoundaryElements** () throw (MEDEXCEPTION)
- void **MEDMEM::SUPPORT::intersecting** (SUPPORT ∗mySupport) throw (MEDEXCEPTION)
- void **MEDMEM::SUPPORT::clearDataOnNumbers** ()

5.5.1 Function Documentation

void blending (SUPPORT ∗ *mySupport*) throw (MEDEXCEPTION) `[inherited]`
Blends the given SUPPORT mySupport into the calling object SUPPORT. Example :

```
SUPPORT mySupport ;
SUPPORT myOtherSupport ;
...
mySupport.blending(myOtherSupport) ;
```

Support *mySupport* now contains a union of the elements originally contained in *mySupport* and *myOtherSupport*.

void getBoundaryElements () throw (MEDEXCEPTION) `[inherited]`
This method gets the boundary elements of the mesh. The support has to be build using the constructor SUPPORT(MESH ∗,string, medEntityMesh) or SUPPORT() followed by setMesh(MESH∗) setName(string) and setEntity(medEntityMesh) before using this method.

void intersecting (SUPPORT ∗ *mySupport*) throw (MEDEXCEPTION) `[inherited]`
Intersects *mySupport* into the calling SUPPORT object. If A.intersecting(B) is called, on output, A contains $A B$.

void clearDataOnNumbers () `[inherited]`
Method that cleans up all the fields related to _numbers. Defined for code factorization.

5.6 Case of FAMILY object

The FAMILY concept is directly linked to the representation of supports in the MED file. It is only useful for directly manipulating the arrays that are written/read by the MED drivers. More information can be found on this topic in MED reference guide [1].

A FAMILY is a SUPPORT with some additional methods that concern some optional attributes (we could have none) and groups (we could also have none) :

- method **getIdentifier** returns the family identifier (an integer)
- method **getNumberOfAttributes** returns the number of attributes of this family
- method **getAttributesIdentifiers** and method **getAttributeIdentifier** returns an integer array or an integer that represents attribute identifier.
- method **getAttributesValues** and method **getAttributeValue** returns an integer array or an integer that represents attribute value.
- method **getAttributesDescriptions** and method **getAttributeDescription** returns a string array or a string that represents attribute description.
- method **getNumberOfGroups** returns the number of groups which it belongs to.
- method **getGroupsNames** and method **getGroupName** return a string array or a string that represents the group name which it belongs to.

5.7 Case of GROUP object

A GROUP is a SUPPORT with some additional methods to find FAMILY that makes it up :

- method **getNumberOfFamilies** returns the number of FAMILY that makes up the GROUP ;
- method **getFamilies** and method **getFamily** return a FAMILY array or a FAMILY that makes up the GROUP.

6 FIELD

6.1 Introduction

MEDMEM fields are used to represent variables over a particular set of elements of the mesh. The region on which the variable is defined is determined through a support object (which can be retrieved by **getSupport()** method). Each field has a number of components, that could for instance be the different coordinates of a vector. All these components have a name, a description and a unit. Elements can also contain several Gauss points, in which case, values are defined on each Gauss point of the element.

The fields can contain integer values or floating point values. In C++, this is reflected by the fact that FIELD is a class template that can be either a `FIELD<int>` or `FIELD<double>`. In Python, two classes `FIELDINT` and `FIELDDOUBLE` exist. In the present section, the methods of the FIELD template will be described as methods of a class `FIELD_` (from which the template classes actually inherit). The template parameter is `T`.

In MEDMEM, a field is characterized by its name (method **getName**) and an optional description (method **getDescription**).

It is also characterized by its computation time :

- an iteration number (time step number)
- an order number (used if there are internal iterations inside a time step)
- the time that corresponds to this iteration number.

By default, there are no iteration and order number defined (value MED_NOPDT and MED_NONOR).

6.2 Interlacing modes

As for the coordinates in the mesh definition, there are two ways to store fields : one consists in interlacing the different components, grouping the data elementwise (MED_FULL_INTERLACE mode), the other one consists in grouping the data componentwise (MED_NO_INTERLACE).

The situation is further complicated by the introduction of Gauss points. If the field is defined on several Gauss points, the MEDMEM convention is that the Gauss points are always grouped together. Let us denote V_{ijk} the value of the field on the i-th element, for the j-th component on its k-th Gauss point. In `MED_FULL_INTERLACE`, elements are nested in a ijk order, while in `MED_NO_INTERLACE` elements are nested in jik order.

For instance, `MED_FULL_INTERLACE` will result in the following ordering (for four Gauss points and two components):

$$V_{111}V_{112}V_{113}V_{114}V_{121}V_{122}V_{123}V_{124}V_{211}V_{212}...$$

`MED_NO_INTERLACE` will result in the following ordering :

$$V_{111}V_{112}V_{113}V_{114}V_{211}V_{212}V_{213}V_{214}V_{311}V_{312}...V_{121}V_{122}V_{123}$$

In this document, only the methods enabling the retrieval of values on fields defined on several Gauss points are presented. For further information on defining the location of the Gauss points in a reference element, the reader should consult document [2]

6.3 Constructors

Functions

- **MEDMEM::FIELD_::FIELD_** (const SUPPORT ∗Support, const int NumberOfComponents)
- **MEDMEM::FIELD_::FIELD_** (const FIELD_ &m)

6.3.1 Detailed Description

These methods define constructors for the fields.

6.3.2 Function Documentation

FIELD_ (const SUPPORT ∗ *Support*, const int *NumberOfComponents*) `[inherited]`
Creating a field from a support.

Parameters:

Support support on which the field is defined

NumberOfComponents number of components for the variable represented in the field

A *double* field representing a 3D-vector defined on face region support *face* is created by :

```
FIELD<double> vector_field(&face, 3);
```

FIELD_ (const FIELD_ & *m*) `[inherited]`
Copy constructor

6.4 Query and Setting methods

Functions

- void **MEDMEM::FIELD_::setName** (const string Name)
- string **MEDMEM::FIELD_::getName** () const
- void **MEDMEM::FIELD_::setDescription** (const string Description)
- string **MEDMEM::FIELD_::getDescription** () const
- void **MEDMEM::FIELD_::setNumberOfComponents** (const int NumberOfComponents)
- int **MEDMEM::FIELD_::getNumberOfComponents** () const

- void **MEDMEM::FIELD_::setNumberOfValues** (const int NumberOfValues)
- int **MEDMEM::FIELD_::getNumberOfValues** () const
- void **MEDMEM::FIELD_::setComponentsNames** (const string ∗ComponentsNames)
- const string ∗ **MEDMEM::FIELD_::getComponentsNames** () const
- void **MEDMEM::FIELD_::setComponentsDescriptions** (const string ∗ComponentsDescriptions)
- const string ∗ **MEDMEM::FIELD_::getComponentsDescriptions** () const
- void **MEDMEM::FIELD_::setMEDComponentsUnits** (const string ∗MEDComponentsUnits)
- const string ∗ **MEDMEM::FIELD_::getMEDComponentsUnits** () const
- void **MEDMEM::FIELD_::setIterationNumber** (int IterationNumber)
- int **MEDMEM::FIELD_::getIterationNumber** () const
- void **MEDMEM::FIELD_::setTime** (double Time)
- double **MEDMEM::FIELD_::getTime** () const
- void **MEDMEM::FIELD_::setOrderNumber** (int OrderNumber)
- int **MEDMEM::FIELD_::getOrderNumber** () const
- const SUPPORT ∗ **MEDMEM::FIELD_::getSupport** () const
- void **MEDMEM::FIELD_::setSupport** (const SUPPORT ∗support)
- MED_EN::med_type_champ **MEDMEM::FIELD_::getValueType** () const
- MED_EN::medModeSwitch **MEDMEM::FIELD_::getInterlacingType** () const

6.4.1 Detailed Description

These methods enable the user to retrieve and set the values of fields.

6.4.2 Function Documentation

void setName (const string *Name*) `[inline, inherited]`
Sets FIELD name. The length should not exceed MED_TAILLE_NOM as defined in Med (i.e. 32 characters).

string getName () const `[inline, inherited]`
Gets FIELD name.

void setDescription (const string *Description*) `[inline, inherited]`
Sets FIELD description. The length should not exceed MED_TAILLE_DESC as defined in Med (i.e. 200 characters).

string getDescription () const `[inline, inherited]`
Gets FIELD description.

void setNumberOfComponents (const int *NumberOfComponents*) `[inline, inherited]`
Sets FIELD number of components.

int getNumberOfComponents () const `[inline, inherited]`
Gets FIELD number of components.

void setNumberOfValues (const int *NumberOfValues*) `[inline, inherited]`
Sets FIELD number of values.

It must be the same than in the associated SUPPORT object.

int getNumberOfValues () const `[inline, inherited]`
Gets FIELD number of value.

void setComponentsNames (const string ∗ *ComponentsNames*) `[inline, inherited]`
Sets FIELD components names.

Duplicates the ComponentsNames string array to put components names in FIELD. ComponentsNames size must be equal to number of components.

const string ∗ getComponentsNames () const `[inline, inherited]`
Gets a reference to the string array which contain the components names.

This Array size is equal to number of components

void setComponentsDescriptions (const string ∗ *ComponentsDescriptions*) `[inline, inherited]`
Sets FIELD components descriptions.

Duplicates the ComponentsDescriptions string array to put components descriptions in FIELD. ComponentsDescriptions size must be equal to number of components.

const string ∗ getComponentsDescriptions () const `[inline, inherited]`
Gets a reference to the string array which contain the components descriptions.

This Array size is equal to number of components

void setMEDComponentsUnits (const string ∗ *MEDComponentsUnits*) `[inline, inherited]`
Sets FIELD components unit.

Duplicates the MEDComponentsUnits string array to put components units in FIELD. MEDComponentsUnits size must be equal to number of components.

const string ∗ getMEDComponentsUnits () const `[inline, inherited]`
Gets a reference to the string array which contain the components units.
This array size is equal to number of components

void setIterationNumber (int *IterationNumber*) `[inline, inherited]`
Sets the iteration number where FIELD has been calculated.

int getIterationNumber () const `[inline, inherited]`
Gets the iteration number where FIELD has been calculated.

void setTime (double *Time*) `[inline, inherited]`
Sets the time when FIELD has been calculated.

double getTime () const `[inline, inherited]`
Gets the time when FIELD has been calculated.

void setOrderNumber (int *OrderNumber*) `[inline, inherited]`
Sets the order number where FIELD has been calculated.
It corresponds to internal iteration during one time step.

int getOrderNumber () const `[inline, inherited]`
Gets the order number where FIELD has been calculated.

const SUPPORT ∗ getSupport () const `[inline, inherited]`
Gets a reference to the SUPPORT object associated to FIELD.

void setSupport (const SUPPORT ∗ *support*) `[inline, inherited]`
Sets the reference to the SUPPORT object associated to FIELD.
Reference is not duplicate, so it must not be deleted.

MED_EN::med_type_champ getValueType () const `[inline, inherited]`
Gets the FIELD med value type (MED_INT32 or MED_REEL64).

MED_EN::medModeSwitch getInterlacingType () const `[inline, inherited]`
Gets the FIELD med interlacing type (MED_FULL_INTERLACE or MED_NO_INTERLACE).

6.5 Field value access methods

Functions

- const T * **MEDMEM::FIELD::getValue** () const throw (MEDEXCEPTION)
- T **MEDMEM::FIELD::getValueIJ** (int i, int j) const throw (MEDEXCEPTION)
- T **MEDMEM::FIELD::getValueIJK** (int i, int j, int k) const throw (MEDEXCEPTION)
- void **MEDMEM::FIELD::setValue** (T *value) throw (MEDEXCEPTION)
- void **MEDMEM::FIELD::setValueIJ** (int i, int j, T value) throw (MEDEXCEPTION)

6.5.1 Detailed Description

These methods enable the user to retrieve or define the values of the field. They access either a specific element of the value array or the whole array, the order having been defined by the interlace mode.

6.5.2 Function Documentation

const T* getValue () const throw (MEDEXCEPTION) `[inline, inherited]`
Returns a pointer to the value array.

T getValueIJ (int *i*, int *j*) const throw (MEDEXCEPTION) `[inline, inherited]`
Returns the value of i^{th} element and j^{th} component. This method only works with fields having no particular Gauss point definition (i.e., fields having one value per element). This method makes the retrieval of the value independent from the interlacing pattern, but it is slower than the complete retrieval obtained by the **getValue()** method.

T getValueIJK (int *i*, int *j*, int *k*) const throw (MEDEXCEPTION) `[inline, inherited]`
Returns the j^{th} component of k^{th} Gauss points of i^{th} value. This method is compatible with elements having more than one Gauss point. This method makes the retrieval of the value independent from the interlacing pattern, but it is slower than the complete retrieval obtained by the **getValue()** method.

void setValue (T * *value*) throw (MEDEXCEPTION) `[inline, inherited]`
Returns the j^{th} component of k^{th} Gauss points of i^{th} value. This method is compatible with elements having more than one Gauss point. This method makes the retrieval of the value independent from the interlacing pattern, but it is slower than the complete retrieval obtained by the **getValue()** method.

void setValueIJ (int *i*, int *j*, T *value*) throw (MEDEXCEPTION) `[inline, inherited]`
Sets the value of i^{th} element and j^{th} component with *value*.

6.5.3 Example for field creation

```
// Copyright (C) 2005  OPEN CASCADE, EADS/CCR, LIP6, CEA/DEN,
// CEDRAT, EDF R&D, LEG, PRINCIPIA R&D, BUREAU VERITAS
//
using namespace std;
#include "MEDMEM_Mesh.hxx"
#include "MEDMEM_Field.hxx"

using namespace MEDMEM;
using namespace MED_EN ;

int main (int argc, char ** argv) {

  const string MedFile = "pointe.med" ;
  const string MeshName = "maa1" ;

  /* read MESH */
  MESH * myMesh = new MESH(MED_DRIVER,MedFile,MeshName) ;
  //  myMesh->read() ;

  // we need a support :
  SUPPORT * mySupport = new SUPPORT(myMesh,"Support on all CELLs",MED_CELL);

  /* create FIELD on mySupport, with 3 components */
  int NumberOfCompoennts = 3 ;
  FIELD<double> myField(mySupport,NumberOfCompoennts) ;
  const string FieldName = "fieldcelldouble" ;
  myField.setName(FieldName) ;

  // Components information
  string * ComponentsNames = new string[NumberOfCompoennts] ;
  ComponentsNames[0] = "Vx" ;
  ComponentsNames[1] = "Vy" ;
  ComponentsNames[2] = "Vz" ;
  myField.setComponentsNames(ComponentsNames) ;

  string * ComponentsDescriptions = new string[NumberOfCompoennts] ;
  ComponentsDescriptions[0] = "vitesse selon x" ;
  ComponentsDescriptions[1] = "vitesse selon y" ;
  ComponentsDescriptions[2] = "vitesse selon z" ;
  myField.setComponentsDescriptions(ComponentsDescriptions) ;

  string * ComponentsUnits = new string[NumberOfCompoennts] ;
  ComponentsUnits[0] = "m.s-1" ;
  ComponentsUnits[1] = "m.s-1" ;
  ComponentsUnits[2] = "m.s-1" ;
  myField.setMEDComponentsUnits(ComponentsUnits) ;

  // Iteration information :
  int IterationNumber = 10 ; // set value to MED_NOPDT if undefined (default)
  myField.setIterationNumber(IterationNumber) ;

  int OrderNumber = 1 ; // set value to MED_NONOR if undefined (default)
  myField.setOrderNumber(OrderNumber) ;

  double Time = 3.435678 ; // in second
  myField.setTime(Time) ;

  // Value :
  int NumberOfValue = mySupport->getNumberOfElements(MED_ALL_ELEMENTS);
  for(int i=1; i<=NumberOfValue; i++) // i^th element
    for (int j=1; j<=NumberOfCompoennts; j++) { // j^th component
      double myValue = (i+j) * 0.1 ;
      myField.setValueIJ(i,j,myValue);
    }

  // save this new field
  int id = myField.addDriver(MED_DRIVER) ;

  return 0 ;
}
```

6.6 File IO methods

Functions

- virtual int **MEDMEM::FIELD_::addDriver** (driverTypes driverType, const string &fileName="Default File Name.med", const string &driverFieldName="Default Field Name", MED_EN::med_mode_acces access=MED_EN::MED_REMP)
- virtual void **MEDMEM::FIELD_::write** (int index=0, const string &driverName="")

6.6.1 Detailed Description

File reading should be done through a FIELD constructor. For file writing, two methods have to be used : one for driver initialization (**addDriver**), the second one triggering the writing of the file (**write**). The MED_DRIVER type is used to specify the format of the input/output file. It can be one of the following values :

- MED_DRIVER,

- VTK_DRIVER (write only).

6.6.2 Function Documentation

int addDriver (driverTypes *driverType*, const string & *fileName* = `"Default File Name.med"`**, const string & *driverFieldName* =** `"Default Field Name"`**, MED_EN::med_mode_acces *access* =** `MED_EN::MED_REMP`**)** `[virtual, inherited]`

Creates a driver for reading/writing fields in a file.

Parameters:

- ***driverType*** specifies the file type (MED_DRIVER, VTK_DRIVER)
- ***fileName*** name of the output file
- ***driverFieldName*** name of the field
- ***access*** specifies whether the file is opened for read, write or both.

void write (int *index* = `0`**, const string & *driverName* =** `""`**)** `[virtual, inherited]`

Triggers the writing of the field with respect to the driver handle *index* given by *addDriver*(...) method.

6.6.3 Example

This program gives an example of creation of a file containing a mesh and fields. This program is a tool that reads a mesh in an input file, creates a field with the inverse of the cell volume, and creates an output file with the mesh and the field.

The reader should note that the mesh name passed as an argument to the `addDriver()` method has to be coherent with the mesh name (as obtained by `getName()`).

```
#include "MEDMEM_Field.hxx"
#include <stdlib.h>

using namespace MEDMEM;
using namespace MED_EN;
using namespace std;

int main(int argc, char ** argv)
{
if (argc != 4)
{
cerr << "Usage : " << argv[0]
<< " inMedfile Meshname outMedfile " << endl << endl
<< "-> Creation of a field the value of which is the inverse of the volume of the cell" <<
exit(-1);
}
const string InputMedFile  = argv[1];
const string Meshname = argv[2];
const string OutputMedFile  = argv[3];
cout<<"create fields : "<< endl << flush;
const int NumberOfComponents=1;
MEDMEM::MESH mesh(MEDMEM::MED_DRIVER,InputMedFile,Meshname1);
SUPPORT support(&mesh,"allCells",MED_CELL);
FIELD<double> *field=mesh.getVolume(&support);
field->setTime(0.);
field->setIterationNumber(1);
field->setOrderNumber(1) ;

// saving the mesh in the output  MED file
int id1=mesh.addDriver(MED_DRIVER,OutputMedFile,mesh.getName());
mesh.write(id1);
//saving the fields in the output MED file
int id2=field->addDriver(MED_DRIVER,OutputMedFile,"Volume");
field->write(id2);

delete field1;
}
```

6.7 Example

The following example reviews most of the notions seen in this section.

```
// Copyright (C) 2005  OPEN CASCADE, EADS/CCR, LIP6, CEA/DEN,
// CEDRAT, EDF R&D, LEG, PRINCIPIA R&D, BUREAU VERITAS
//
using namespace std;
#include "MEDMEM_Mesh.hxx"
#include "MEDMEM_Field.hxx"

using namespace MEDMEM;
using namespace MED_EN ;

int main (int argc, char ** argv) {

  const string MedFile = "pointe.med" ;
  const string MeshName = "maa1" ;
  const string FieldName = "fieldcelldoublevector" ;

  /* read MESH */
  MESH * myMesh = new MESH(MED_DRIVER,MedFile,MeshName) ;
  //  myMesh->read() ;

  /* read FIELD */
  // we need a support :
  SUPPORT * mySupport = new SUPPORT(myMesh,"Support on all Cells",MED_CELL);
  FIELD<double> myField(mySupport,MED_DRIVER,MedFile,FieldName) ;
  //  myField.read() ;

  /* what in Field ? */
  // How many components
  int NumberOfCompoennts = myField.getNumberOfComponents() ;

  const string * ComponentsNames = myField.getComponentsNames();
  const string * ComponentsDescriptions = myField.getComponentsDescriptions();
  const string * ComponentsUnits = myField.getMEDComponentsUnits();

  for(int i=0;i<NumberOfCompoennts; i++) {
    cout << "Component " << i << " :" <<endl ;
    cout << "  - name        : " << ComponentsNames[i] << endl ;
    cout << "  - description : " << ComponentsDescriptions[i] << endl ;
    cout << "  - unit        : " << ComponentsUnits[i] << endl ;
  }

  // Which iteration :
  int IterationNumber = myField.getIterationNumber() ; // negative mean undefined
  int OrderNumber = myField.getOrderNumber() ;
  // internal iteration at this time iteration, negative mean undefined
  double Time = myField.getTime() ;

  cout << "Iteration " << IterationNumber << " at time " << Time <<
    " (and order number " << OrderNumber << ")" << endl ;

  // How many Value :
  int NumberOfValue = mySupport->getNumberOfElements(MED_ALL_ELEMENTS);
  // Value
  const double * Value = myField.getValue();
  for(int i=0; i<NumberOfValue; i++) {
    for(int j=0; j<NumberOfCompoennts; j++)
      cout << Value[i*NumberOfCompoennts+j] << " " ;
    cout << endl ;
  }

  delete mySupport;
  delete myMesh;

  return 0 ;
}
```

7 MED object

7.1 General Information

This object is used to give information about the different meshes/supports/fields that are contained in a file. This enables the user to know about the file content without loading the meshes in memory. Also, it can be useful for memory management since meshes, supports and fields accessed through a MED object are destroyed when the MED object is destroyed.

7.2 Constructors

Functions

- **MEDMEM::MED::MED** (driverTypes driverType, const string &fileName)

7.2.1 Function Documentation

MED (driverTypes *driverType*, const string & *fileName*) `[inherited]`
This constructor constructs the Med object from the file *filename*. The driver type can specify whether the file is opened in read, write or read/write mode. Specifying MED_DRIVER as a *driverType* opens the file in read/write mode. It is also possible to use VTK_DRIVER to open a VTK ascii file.

7.3 Query methods

Functions

- int **MEDMEM::MED::getNumberOfMeshes** (void) const
- int **MEDMEM::MED::getNumberOfFields** (void) const
- void **MEDMEM::MED::getMeshNames** (string ∗meshNames) const throw (MEDEXCEPTION)
- deque< string > **MEDMEM::MED::getMeshNames** () const
- MESH ∗ **MEDMEM::MED::getMesh** (const string &meshName) const throw (MEDEXCEPTION)
- void **MEDMEM::MED::getFieldNames** (string ∗fieldNames) const throw (MEDEXCEPTION)
- deque< string > **MEDMEM::MED::getFieldNames** () const
- deque< DT_IT_ > **MEDMEM::MED::getFieldIteration** (const string &fieldName) const throw (MEDEXCEPTION)
- FIELD_ ∗ **MEDMEM::MED::getField** (const string &fieldName, const int dt, const int it) const throw (MEDEXCEPTION)
- FIELD_ ∗ **MEDMEM::MED::getField2** (const string &fieldName, double time, int it=0) const throw (MEDEXCEPTION)
- SUPPORT ∗ **MEDMEM::MED::getSupport** (const string &meshName, MED_EN::medEntityMesh entity) const throw (MEDEXCEPTION)
- void **MEDMEM::MED::updateSupport** ()

7.3.1 Detailed Description

These methods enable the user to retrieve information about a MED file structure, i.e. the meshes, supports and fields that it contains.

7.3.2 Function Documentation

int getNumberOfMeshes (void) const `[inherited]`
Gets the number of MESH objects.

int getNumberOfFields (void) const `[inherited]`
Gets the number of FIELD objects.

void getMeshNames (string ∗ *meshNames*) const throw (MEDEXCEPTION) `[inherited]`
Gets the names of all MESH objects.

meshNames is an in/out argument.

It is a string array of size the number of MESH objects. It must be allocated before calling this method. All names are put in it.

deque< string > getMeshNames () const `[inherited]`
Gets the names of all MESH objects.

Returns a deque<string> object which contain the name of all MESH objects.

MESH ∗ getMesh (const string & *meshName*) const throw (MEDEXCEPTION) `[inherited]`
Returns a reference to the MESH object named meshName.

void getFieldNames (string ∗ *fieldNames*) const throw (MEDEXCEPTION) `[inherited]`
Gets the names of all FIELD objects.

fieldNames is an in/out argument.

It is an array of string of size the number of FIELD objects. It must be allocated before calling this method. All names are put in it.

deque< string > getFieldNames () const `[inherited]`
Gets the names of all FIELD objects.

Returns a deque<string> object which contain the name of all FIELD objects.

deque< DT_IT_ > getFieldIteration (const string & *fieldName*) const throw (MEDEXCEPTION) `[inherited]`

Returns a deque<DT_IT_> which contain all iteration step for the FIELD identified by its name. DT_IT_ definition is

```
typedef struct { int dt; int it; } DT_IT_;
```

dt represents the time iteration number, while *it* represents the inner iteration number.

FIELD_ ∗ getField (const string & *fieldName*, const int *dt* = `MED_NOPDT`**, const int *it* =** `MED_NOPDT`**) const throw (MEDEXCEPTION)** `[inherited]`

Returns a reference to the FIELD object named fieldName with time step number dt and order number it.

FIELD_ ∗ getField2 (const string & *fieldName*, double *time*, int *it* = `0`**) const throw (MEDEXCEPTION)** `[inherited]`

Returns a reference to the FIELD object named fieldName with time and iteration nb it.

SUPPORT ∗ getSupport (const string & *meshName*, MED_EN::medEntityMesh *entity*) const throw (MEDEXCEPTION) `[inherited]`

Returns a reference to the SUPPORT object on all elements of entity for the MESH named meshName.

void updateSupport () `[inherited]`

The need for this method arises from the following situation. When loading a mesh, Medmem reads the constituent elements in the Med file. It is possible at this stage that not all constituent elements are stored in memory (For instance, reading a 2D mesh, not all the edges are present, because only the boundaries were stored in the file). When computing descending connectivities, Medmem stores all the faces and has a corresponding numbering. This introduces a discrepancy between the support numbering which was determined at file loading and the new numbering. The following method synchronizes the two numberings.

For an example using these methods, one may see the Python scripts in the directory `$MED_ROOT_DIR/bin/salome/`,`testMedObj.py`, or C++ example program in the directory `$MED_SRC_DIR/src/MEDMEM`, `duplicateMED.cxx`.

8 MESHING

This class is a class derived from MESH class that is used to build a MESH object from scratch.
All verifications are under user responsability : if array values or array dimensions are wrong, results are impredictable. All the arrays passed as arguments in the set methods are duplicated in MESHING object.
The creation of a mesh should respect the following sequence :

1. setting general information (name, description, dimensions, coordinate system, ...)
2. setting the nodes (number and coordinates)
3. setting the connectivity (types, connectivity arrays,...)
4. group creations

The following paragraphs describe the methods that must be called when creating a mesh. An example illustrates the general procedure. The specific case of `MED_POLYGON` and `MED_POLYHEDRA` elements requires some methods that are described in 9.

8.1 Constructors

Functions

- **MEDMEM::MESHING::MESHING** ()

8.1.1 Function Documentation

MESHING () `[inherited]`
Creates an empty MESH.

8.2 General information settings

Functions

- void **MEDMEM::MESHING::setSpaceDimension** (const int SpaceDimension)
- void **MEDMEM::MESHING::setMeshDimension** (const int MeshDimension)
- void **MEDMEM::MESHING::setCoordinatesNames** (const string ∗names)
- void **MEDMEM::MESHING::setCoordinatesUnits** (const string ∗units)

8.2.1 Function Documentation

void setSpaceDimension (const int *SpaceDimension*) `[inherited]`
Sets the dimension of the space.

void setMeshDimension (const int *MeshDimension*) `[inherited]`
Sets the dimension of the mesh.

void setCoordinatesNames (const string ∗ *name*) `[inherited]`
Sets the coordinate names array. Coordinates names must not exceed the storage length defined in MED-file : MED_TAILLE_PNOM (8).
Example:

```
string coord[3]={"x","y","z"};
meshing.setCoordinatesNames(coord);
```

void setCoordinatesUnits (const string ∗ *units*) `[inherited]`
Set the coordinate unit names array of size n∗MED_TAILLE_PNOM. Coordinates units must not exceed the storage length defined in MED-file : MED_TAILLE_PNOM (8).
Example:

```
string coord[3]={"cm","cm","cm"};
meshing.setCoordinatesUnits(coord);
```

8.3 Node coordinates settings

Functions

- void **MEDMEM::MESHING::setNumberOfNodes** (const int NumberOfNodes)
- void **MEDMEM::MESHING::setCoordinates** (const int SpaceDimension, const int NumberOfNodes, const double ∗Coordinates, const string System, const MED_EN::medModeSwitch Mode)

8.3.1 Function Documentation

void setNumberOfNodes (const int *NumberOfNodes*) `[inherited]`
Sets the number of nodes used in the mesh.

void setCoordinates (const int *SpaceDimension*, const int *NumberOfNodes*, const double ∗ *Coordinates*, const string *System*, const MED_EN::medModeSwitch *Mode*) `[inherited]`
Sets the whole coordinates array in a given system and interlacing mode. The system coordinates are :

- "MED_CART"
- "MED_CYL"
- "MED_SPHER" The interlacing mode are :

- MED_NO_INTERLACE : X1 X2 Y1 Y2 Z1 Z2
- MED_FULL_INTERLACE : X1 Y1 Z1 X2 Y2 Z2

Example :

```
MESHING myMeshing ;
const int SpaceDimension=2;
const int NumberOfNodes=6;
int * Coordinates = new int[SpaceDimension*NumberOfNodes] ;
string System="CARTESIAN";
medModeSwitch Mode = MED_FULL_INTERLACE ;
myMeshing.setCoordinates(SpaceDimension,NumberOfNodes,Coordinates,System,Mode);
```

8.4 Connectivity settings

Functions

- void **MEDMEM::MESHING::setNumberOfTypes** (const int NumberOfTypes, const MED_EN::medEntityMesh Entity) throw (MEDEXCEPTION)
- void **MEDMEM::MESHING::setTypes** (const MED_EN::medGeometryElement *Types, const MED_EN::medEntityMesh Entity) throw (MEDEXCEPTION)
- void **MEDMEM::MESHING::setNumberOfElements** (const int *NumberOfElements, const MED_EN::medEntityMesh Entity) throw (MEDEXCEPTION)
- void **MEDMEM::MESHING::setConnectivity** (const int *Connectivity, const MED_EN::medEntityMesh Entity, const MED_EN::medGeometryElement Type) throw (MEDEXCEPTION)

8.4.1 Detailed Description

When defining the connectivity, MED_CELL elements connectivity should be defined first. If necessary, constituent connectivities (MED_FACE and/or MED_EDGE) can be defined afterwards.

Warning:

it should be kept in mind that when defining connectivities, elements should be sorted in ascending type order (the type order being defined by the number of nodes).

8.4.2 Function Documentation

void setNumberOfTypes (const int *NumberOfTypes*, const MED_EN::medEntityMesh *Entity*) throw (MEDEXCEPTION) `[inherited]`

Creates a new connectivity object with the given number of type and entity. If a connectivity already exist, it is deleted by the call.

For exemple setNumberOfTypes(3,MED_CELL) creates a connectivity with 3 medGeometryElement in MESH for MED_CELL entity (like MED_TETRA4, MED_PYRA5 and MED_HEXA6 for example).

Returns an exception if it could not create the connectivity (as if we set MED_FACE connectivity before MED_CELL).

void setTypes (const MED_EN::medGeometryElement ∗ *Types*, const MED_EN::medEntityMesh *Entity*) throw (MEDEXCEPTION) `[inherited]`

Sets the list of geometric types used by a given entity. medEntityMesh entity could be : MED_CELL, MED_FACE, MED_EDGE. This method is used to set the differents geometrics types ({MED_TETRA4,MED_PYRA5,MED_HEXA8} for example). Geometric types should be given in increasing order of number of nodes for entity type *entity*.

Remark : Don't use MED_NODE and MED_ALL_ENTITIES.

If *entity* is not defined, the method will throw an exception.

void setNumberOfElements (const int ∗ *NumberOfElements*, const MED_EN::medEntityMesh *Entity*) throw (MEDEXCEPTION) `[inherited]`

Sets the number of elements for each geometric type of given entity.

Example : setNumberOfElements(12,23,MED_FACE); If there are two types of face (MED_TRIA3 and MED_QUAD4), this sets 12 triangles and 23 quadrangles.

void setConnectivity (const int ∗ *Connectivity*, const MED_EN::medEntityMesh *Entity*, const MED_EN::medGeometryElement *Type*) throw (MEDEXCEPTION) `[inherited]`

Sets the nodal connectivity for geometric type *Type* of entity *Entity*. The nodal connectivity must be defined one element type at a time : *MED_ALL_ELEMENTS* is not a valid *Type* argument.

Example :

```
MESHING myMeshing ;
myMeshing.setCoordinates(SpaceDimension,NumberOfNodes,Coordinates,System,Mode);

myMeshing.setNumberOfTypes(2,MED_CELL);
myMeshing.setTypes({MED_TRIA3,MED_QUAD4},MED_CELL);
myMeshing.setNumberOfElements({3,2},MED_CELL); // 3 MED_TRIA3 and 2 MED_QUAD4
myMeshing.setConnectivity({1,2,3,6,8,9,4,5,6},MED_CELL,MED_TRIA3);
myMeshing.setConnectivity({1,3,4,5,4,5,7,8},MED_CELL,MED_QUAD4);
```

8.5 Group creation

Functions

- void **MEDMEM::MESHING::addGroup** (const GROUP &Group) throw (MEDEXCEPTION)

8.5.1 Function Documentation

void addGroup (const GROUP & *Group*) throw (MEDEXCEPTION) `[inherited]`

Adds group *Group* to the mesh. This function registers the group in the list of groups contained in the mesh, so that when the mesh is used for file writing, the group is written in the corresponding MED-file.

8.6 Full C++ example :

```
// Copyright (C) 2005  OPEN CASCADE, EADS/CCR, LIP6, CEA/DEN,
// CEDRAT, EDF R&D, LEG, PRINCIPIA R&D, BUREAU VERITAS
//
#include "MEDMEM_Meshing.hxx"
#include "MEDMEM_Group.hxx"

using namespace MEDMEM ;
using namespace MED_EN ;

using namespace std;

int main (int argc, char ** argv) {

  // filename to save the generated MESH
  string filename = "meshing.med" ;

  MESHING myMeshing ;
  myMeshing.setName("meshing") ;

  // define coordinates

  int SpaceDimension = 3 ;
  int MeshDimension = 3;
  int NumberOfNodes = 19 ;
  double Coordinates[57] = {
    0.0, 0.0, 0.0,
    0.0, 0.0, 1.0,
    2.0, 0.0, 1.0,
    0.0, 2.0, 1.0,
    -2.0, 0.0, 1.0,
    0.0, -2.0, 1.0,
    1.0, 1.0, 2.0,
    -1.0, 1.0, 2.0,
    -1.0, -1.0, 2.0,
    1.0, -1.0, 2.0,
    1.0, 1.0, 3.0,
    -1.0, 1.0, 3.0,
  -1.0, -1.0, 3.0,
  1.0, -1.0, 3.0,
  1.0, 1.0, 4.0,
  -1.0, 1.0, 4.0,
  -1.0, -1.0, 4.0,
  1.0, -1.0, 4.0,
  0.0, 0.0, 5.0
};

myMeshing.setMeshDimension(MeshDimension);

myMeshing.setCoordinates(SpaceDimension,NumberOfNodes,Coordinates,"CARTESIAN",MED

string Names[3] = { "X","Y","Z" } ;
myMeshing.setCoordinatesNames(Names);

string Units[3] = { "cm","cm","cm" } ;
myMeshing.setCoordinatesUnits(Units) ;

// define connectivities

// cell part

const int NumberOfTypes = 3 ;
medGeometryElement Types[NumberOfTypes] = {MED_TETRA4,MED_PYRA5,MED_HEXA8} ;
const int NumberOfElements[NumberOfTypes] = {12,2,2} ;

myMeshing.setNumberOfTypes(NumberOfTypes,MED_CELL);
myMeshing.setTypes(Types,MED_CELL);
myMeshing.setNumberOfElements(NumberOfElements,MED_CELL);

const int sizeTetra = 12*4 ;
int ConnectivityTetra[sizeTetra]=
{
  1,2,3,6,
  1,2,4,3,
  1,2,5,4,
  1,2,6,5,
  2,7,4,3,
  2,8,5,4,
  2,9,6,5,
  2,10,3,6,
  2,7,3,10,
  2,8,4,7,
  2,9,5,8,
  2,10,6,9
};

myMeshing.setConnectivity(ConnectivityTetra,MED_CELL,MED_TETRA4);

int ConnectivityPyra[2*5]=
{
  7,8,9,10,2,
  15,18,17,16,19
};

myMeshing.setConnectivity(ConnectivityPyra,MED_CELL,MED_PYRA5);

int ConnectivityHexa[2*8]=
{
  11,12,13,14,7,8,9,10,
  15,16,17,18,11,12,13,14
};

myMeshing.setConnectivity(ConnectivityHexa,MED_CELL,MED_HEXA8);

// face part

const int NumberOfFacesTypes = 2 ;
medGeometryElement FacesTypes[NumberOfFacesTypes] = {MED_TRIA3,MED_QUAD4} ;
const int NumberOfFacesElements[NumberOfFacesTypes] = {4,4} ;

myMeshing.setNumberOfTypes(NumberOfFacesTypes,MED_FACE);
myMeshing.setTypes(FacesTypes,MED_FACE);
myMeshing.setNumberOfElements(NumberOfFacesElements,MED_FACE);

const int sizeTria = 3*4 ;
int ConnectivityTria[sizeTria]=
{
  1,4,3,
  1,5,4,
  1,6,5,
  1,3,6
};
```

```
myMeshing.setConnectivity(ConnectivityTria,MED_FACE,MED_TRIA3);

int ConnectivityQua[4*4]=
{
  7,8,9,10,
  11,12,13,14,
  11,7,8,12,
  12,8,9,13
};

myMeshing.setConnectivity(ConnectivityQua,MED_FACE,MED_QUAD4);

// Some groups :

// Node :
{
  GROUP myGroup ;
  myGroup.setName("SomeNodes");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_NODE);
  myGroup.setNumberOfGeometricType(1);
  medGeometryElement myTypes[1] = {MED_NONE};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[1] = {4} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[1+1] = {1,5} ;
  const int value[4]= { 1,4,5,7} ;
  myGroup.setNumber(index,value);

  myMeshing.addGroup(myGroup);
}
{
  GROUP myGroup ;
  myGroup.setName("OtherNodes");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_NODE);
  myGroup.setNumberOfGeometricType(1);
  medGeometryElement myTypes[1] = {MED_NONE};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[1] = {3} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[1+1] = {1,4} ;
  const int value[3]= { 2,3,6} ;
  myGroup.setNumber(index,value);

  myMeshing.addGroup(myGroup);
}

// Cell :
{
  GROUP myGroup ;
  myGroup.setName("SomeCells");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_CELL);
  myGroup.setNumberOfGeometricType(3);
  medGeometryElement myTypes[3] = {MED_TETRA4,MED_PYRA5,MED_HEXA8};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[3] = {4,1,2} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[3+1] = {1,5,6,8} ;
  const int value[4+1+2]=
  {
    2,7,8,12,
    13,
    15,16
  };
  myGroup.setNumber(index,value);

  myMeshing.addGroup(myGroup);
}
{
  GROUP myGroup ;
  myGroup.setName("OtherCells");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_CELL);
  myGroup.setNumberOfGeometricType(2);
  medGeometryElement myTypes[] = {MED_TETRA4,MED_PYRA5};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[] = {4,1} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[3+1] = {1,5,6} ;
  const int value[4+1]=
  {
    3,4,5,9,
    14
  };
  myGroup.setNumber(index,value);

  myMeshing.addGroup(myGroup);
}

// Face :
{
  GROUP myGroup ;
  myGroup.setName("SomeFaces");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_FACE);
  myGroup.setNumberOfGeometricType(2);
  medGeometryElement myTypes[2] = {MED_TRIA3,MED_QUAD4};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[2] = {2,3} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[2+1] = {1,3,6} ;
  const int value[2+3]=
  {
    2,4,
    5,6,8
  } ;
  myGroup.setNumber(index,value);

  myMeshing.addGroup(myGroup);
}
{
  GROUP myGroup ;
  myGroup.setName("OtherFaces");
  myGroup.setMesh(&myMeshing);
  myGroup.setEntity(MED_FACE);
  myGroup.setNumberOfGeometricType(1);
  medGeometryElement myTypes[1] = {MED_TRIA3};
  myGroup.setGeometricType(myTypes);
  const int myNumberOfElements[1] = {2} ;
  myGroup.setNumberOfElements(myNumberOfElements);
  const int index[1+1] = {1,3} ;
  const int value[2]=
  {
      1,3
    } ;
    myGroup.setNumber(index,value);

    myMeshing.addGroup(myGroup);
  }

  // all rigtht, we save it !

  int id = myMeshing.addDriver(MED_DRIVER,filename,myMeshing.getName());
  myMeshing.write(id) ;

}
```

The Python equivalent of this example can be found in `MESHINGexample.py`.

9 Polyhedra and Polygons

9.1 General information

The methods described in section 3 do not take into account information about `polygonal` and `polyhedral` cells contained in a MESH object. Indeed, in the MEDMEM library, the connectivity data for these elements are stored separately . Therefore, the methods that give access to this data are slightly different from those of section 3.

Also, the polygon and the polyhedra case differ in nature, because in 3D, the list of nodes is not sufficient to described the shape of an element. A descending cell>face>nodes connectivity has to be established to fully describe the elements.

9.1.1 Polygon connectivity

Let us consider the case illustrated in figure 6.

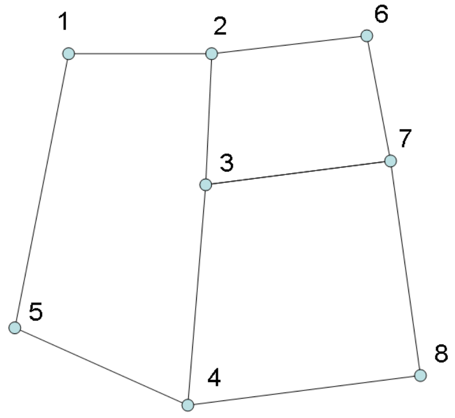


Figure 6: Example for polygon connectivity

The standard element connectivity table writes :
{2, 6, 7, 3, 3, 7, 8, 4}
The standard element connectivity index table writes :
{1, 5, 9}
The polygon element connectivity table writes :

{1, 2, 3, 4, 5}
The polygon element connectivity index table writes :
{1, 6}

9.1.2 Polyhedron connectivity

For polyhedra, in the nodal connectivity case, one more array is required, because a list of nodes does not suffice to describe a general polyhedron. A general polyhedron is therefore described by a list of faces, each of those being described by a list of nodes.

Let us consider an example with the two tetrahedra represented on figure 7 , the left one being stored as a `MED_TETRA4` element, the right one being stored as a `MED_POLYHEDRA` element.

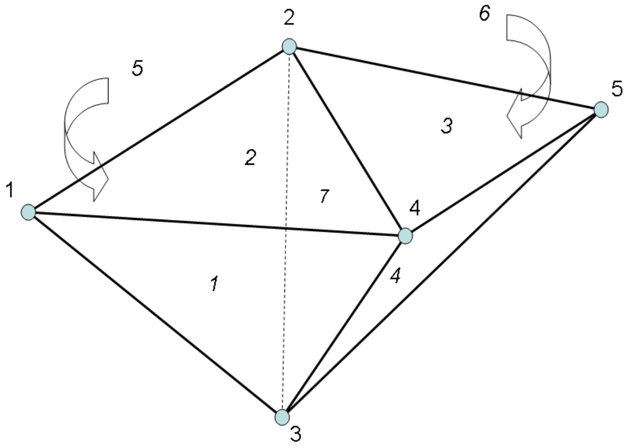


Figure 7: Example for polyhedron connectivity. Node numbers are written with a normal font, while face numbers are written in italic font.

The standard element index connectivity table writes :
{1, 5}
The standard element connectivity table writes :
{1, 2, 3, 4}
The polyhedra face connectivity index table writes :
{1, 5}
The polyhedra connectivity index table writes :

{1, 4, 7, 10, 13}
The polyhedra connectivity (face/node connectivity) table writes :
{2, 3, 5, 2, 4, 5, 4, 5, 3, 2, 3, 4}

Note that as they are not needed as such, the face numberings are not stored in any array. Only the number of nodes per face is implicitly stored in the polyhedra face connectivity index table.

If there are two `MED_POLYHEDRA` elements that share a common face, the list of nodes is repeated twice in the polyhedron connectivity array.

9.2 Polygons and Polyhedra information

Functions

- virtual int **MEDMEM::MESH::getNumberOfTypesWithPoly** (MED_EN::medEntityMesh Entity) const
- virtual MED_EN::medGeometryElement * **MEDMEM::MESH::getTypesWithPoly** (MED_EN::medEntityMesh Entity) const
- virtual int **MEDMEM::MESH::getNumberOfElementsWithPoly** (MED_EN::medEntityMesh Entity, MED_EN::medGeometryElement Type) const
- virtual MED_EN::medGeometryElement **MEDMEM::MESH::getElementTypeWithPoly** (MED_EN::medEntityMesh Entity, int Number) const
- const int * **MEDMEM::MESH::getPolygonsConnectivity** (MED_EN::medConnectivity ConnectivityType, MED_EN::medEntityMesh Entity) const
- const int * **MEDMEM::MESH::getPolygonsConnectivityIndex** (MED_EN::medConnectivity ConnectivityType, MED_EN::medEntityMesh Entity) const
- int **MEDMEM::MESH::getNumberOfPolygons** (MED_EN::medEntityMesh Entity=MED_EN::MED_ALL_ENTITIES) const
- const int * **MEDMEM::MESH::getPolyhedronConnectivity** (MED_EN::medConnectivity ConnectivityType) const
- const int * **MEDMEM::MESH::getPolyhedronFacesIndex** () const
- const int * **MEDMEM::MESH::getPolyhedronIndex** (MED_EN::medConnectivity ConnectivityType) const
- int **MEDMEM::MESH::getNumberOfPolyhedronFaces** () const
- int **MEDMEM::MESH::getNumberOfPolyhedron** () const

9.2.1 Detailed Description

These methods are specific methods used for retrieving connectivity information for `MED_POLYGON` and `MED_POLYHEDRON` elements.

9.2.2 Function Documentation

int getNumberOfTypesWithPoly (MED_EN::medEntityMesh *Entity*) const `[virtual, inherited]`

Method equivalent to getNumberOfTypes except that it includes not only classical Types but polygons/polyhedra also.

MED_EN::medGeometryElement * getTypesWithPoly (MED_EN::medEntityMesh *Entity*) const `[virtual, inherited]`

Method equivalent to getTypesWithPoly except that it includes not only classical Types but polygons/polyhedra also. Memory management of the returned array is under responsibility of the calling code.

int getNumberOfElementsWithPoly (MED_EN::medEntityMesh *Entity*, MED_EN::medGeometryElement *Type*) const `[virtual, inherited]`

Method equivalent to getNumberOfElementsWithPoly except that it includes not only classical Types but polygons/polyhedra also.

MED_EN::medGeometryElement getElementTypeWithPoly (MED_EN::medEntityMesh *Entity*, int *Number*) const `[inline, virtual, inherited]`

Method equivalent to getElementType except that it includes not only classical Types but polygons/polyhedra also.

const int * getPolygonsConnectivity (MED_EN::medConnectivity *ConnectivityType*, MED_EN::medEntityMesh *Entity*) const `[inline, inherited]`

Return the required connectivity of polygons for the given entity. You must also get the corresponding index array.

const int * getPolygonsConnectivityIndex (MED_EN::medConnectivity *ConnectivityType*, MED_EN::medEntityMesh *Entity*) const `[inline, inherited]`

Return the required index array for polygons connectivity.

int getNumberOfPolygons (MED_EN::medEntityMesh *Entity* = `MED_EN::MED_ALL_ENTITIES`**) const** `[inline, inherited]`

Return the number of polygons.

const int * getPolyhedronConnectivity (MED_EN::medConnectivity *ConnectivityType*) const `[inline, inherited]`

Returns the required connectivity of polyhedron :

- in nodal mode, it gives you the polyhedron faces nodal connectivity.
- in descending mode, it gives you the polyhedron faces list.

You must also get :

- faces index and polyhedron index arrays in nodal mode.
- polyhedron index array in descending mode.

const int ∗ getPolyhedronFacesIndex () const `[inline, inherited]`
Returns the index array of polyhedron faces in nodal mode. You must also get the polyhedron index array.

const int ∗ getPolyhedronIndex (MED_EN::medConnectivity *ConnectivityType*) const `[inline, inherited]`
Returns the required polyhedron index array.

int getNumberOfPolyhedronFaces () const `[inline, inherited]`
Returns the number of polyhedron faces.

int getNumberOfPolyhedron () const `[inline, inherited]`
Returns the number of polyhedron.

9.3 Polygons and Polyhedra creation

Functions

- void **MEDMEM::MESHING::setPolyhedraConnectivity** (const int ∗PolyhedronIndex, const int ∗FacesIndex, const int ∗Nodes, int nbOfPolyhedra, const MED_EN::medEntityMesh Entity) throw (MEDEXCEPTION)

9.3.1 Detailed Description

These methods belong to the meshing class and are necessary for creating the connectivities of MED_POLYHEDRON and MED_POLYGON elements.

9.3.2 Function Documentation

void setPolyhedraConnectivity (const int ∗ *PolyhedronIndex*, const int ∗ *FacesIndex*, const int ∗ *Nodes*, int *nbOfPolyhedra*, const MED_EN::medEntityMesh *Entity*) throw (MEDEXCEPTION) `[inherited]`
Method setting the connectivity for MED_POLYHEDRON elements

Parameters:

- ***PolyhedronIndex*** polyhedra connectivity index
- ***FacesIndex*** polyhedra face connectivity index
- ***Nodes*** polyhedra connectivity
- ***nbOfPolyhedra*** number of polyhedra defined
- ***Entity*** deprecated parameter

10 Appendix: Python example scripts

10.1 Full Python example for 3.4.3 :

```
# Copyright (C) 2005  OPEN CASCADE, CEA, EDF R&D, LEG
#            PRINCIPIA R&D, EADS CCR, Lip6, BV, CEDRAT
#
from libMEDMEM_Swig import *

MedFile = "pointe.med"
meshName = "maa1"

myMesh = MESH(MED_DRIVER,MedFile,meshName)

name = myMesh.getName()

if (name != meshName) :
    print "Error when reading mesh name : We ask for mesh #",meshName,"#"
    print "and we get mesh #",name
else :
    print "Mesh name : ",name
    spaceDimension = myMesh.getSpaceDimension()
    meshDimension = myMesh.getMeshDimension()
    print "Space Dimension : ",spaceDimension
    print "Mesh Dimension : ",meshDimension
```

10.2 Full Python example for 3.5.3 :

```
# Copyright (C) 2005  OPEN CASCADE, CEA, EDF R&D, LEG
#            PRINCIPIA R&D, EADS CCR, Lip6, BV, CEDRAT
#
from libMEDMEM_Swig import *

MedFile = "pointe.med"
meshName = "maa1"

myMesh = MESH(MED_DRIVER,MedFile,meshName)

name = myMesh.getName()

print "Mesh name : ",name
spaceDimension = myMesh.getSpaceDimension()
numberOfNodes = myMesh.getNumberOfNodes()
print "Space Dimension : ",spaceDimension
print "Number of Nodes : ",numberOfNodes

print "Show Nodes Coordinates :"
print "Name :"
coordinatesNames = myMesh.getCoordinatesNames()
for i in range(spaceDimension):
    coordinateName = coordinatesNames[i]
    print " - ",coordinateName

print "Unit :"
coordinatesUnits = myMesh.getCoordinatesUnits()
for i in range(spaceDimension):
    coordinateUnit = coordinatesUnits[i]
    print " - ",coordinateUnit

coordinates = myMesh.getCoordinates(MED_FULL_INTERLACE)
for i in range(numberOfNodes):
    print "Node ",(i+1)," : ",coordinates[i*spaceDimension:(i+1)*spaceDimension]
```

10.3 Full Python example for 6.7 :

```
# Copyright (C) 2005  OPEN CASCADE, CEA, EDF R&D, LEG
#           PRINCIPIA R&D, EADS CCR, Lip6, BV, CEDRAT
#
######################################################################
#                                                                    #
# This Python script should be executed when the shared library is   #
# generated using SWIG 1.3 (or higher) due to the fact that older    #
# version could not handle the wrapping of several class constructor #
#                                                                    #
######################################################################
from libMEDMEM_Swig import *

MedFile = "pointe.med"
meshName = "maa1"
fieldName = "fieldcelldouble"

myMesh = MESH(MED_DRIVER,MedFile,meshName)

mySupport = SUPPORT(myMesh,"Support on CELLs",MED_CELL)

myField = FIELDDOUBLE(mySupport,MED_DRIVER,MedFile,fieldName)

numberOfComponents = myField.getNumberOfComponents()

for i in range(numberOfComponents):
    ip1 = i+1
    name = myField.getComponentName(ip1)
    desc = myField.getComponentDescription(ip1)
    unit = myField.getMEDComponentUnit(ip1)
    print "Component ",ip1
    print "  - name        : ",name
    print "  - decription  : ",desc
    print "  - unit        : ", unit

iterationNumber = myField.getIterationNumber()
orderNumber = myField.getOrderNumber()
time = myField.getTime()
print "Iteration ",iterationNumber,"  at time ",time,\
      " (and order number ",orderNumber,")"

numberOfValue = mySupport.getNumberOfElements(MED_ALL_ELEMENTS)
value = myField.getValue(MED_FULL_INTERLACE)

for i in range(numberOfValue):
    print "  * ",value[i*numberOfComponents:(i+1)*numberOfComponents]
```

10.4 Full Python example for 6.5.3 :

```
# Copyright (C) 2005  OPEN CASCADE, CEA, EDF R&D, LEG
#           PRINCIPIA R&D, EADS CCR, Lip6, BV, CEDRAT
#
######################################################################
#                                                                    #
# This Python script should be executed when the shared library is   #
# generated using SWIG 1.3 (or higher) due to the fact that older    #
# version could not handle the wrapping of several class constructor #
#                                                                    #
######################################################################
from libMEDMEM_Swig import *

MedFile = "pointe.med"
meshName = "maa1"

myMesh = MESH(MED_DRIVER,MedFile,meshName)

mySupport = SUPPORT(myMesh,"Support on all CELLs",MED_CELL)

numberOfComponents = 3
myField = FIELDDOUBLE(mySupport,numberOfComponents)
fieldName = "fieldcelldouble"
myField.setName(fieldName)

for i in range(numberOfComponents):
    if (i == 0):
        name = "Vx"
        desc = "vitesse selon x"
    elif (i == 1):
        name = "Vy"
        desc = "vitesse selon y"
    else:
        name = "Vz"
        desc = "vitesse selon z"
    unit = "m. s-1"
    ip1 = i+1
    myField.setComponentName(ip1,name)
    myField.setComponentDescription(ip1,desc)
    myField.setMEDComponentUnit(ip1,unit)

iterationNumber = 10
myField.setIterationNumber(iterationNumber)

orderNumber = 1
myField.setOrderNumber(orderNumber)

time = 3.435678
myField.setTime(time)

numberOfValue = mySupport.getNumberOfElements(MED_ALL_ELEMENTS)

for i in range(numberOfValue):
    ip1 = i+1
    for j in range(numberOfComponents):
        jp1 = j+1
        value = (ip1+jp1)*0.1
        myField.setValueIJ(ip1,jp1,value)

id = myField.addDriver(MED_DRIVER)
```

References

[1] Reference Manual for Med File :
V. LEFEBVRE, E. FAYOLLE
Projet PAL: Définition du modèle d'échange de données MED V2.2 *Note technique EDF/SINETICS* HI-26-03-012/A
`https://hammi.extra.cea.fr/static/MED/web_med/index.html`

[2] Med Memory Users Reference Manual :
`file:://$MED_ROOT_DIR/share/salome/doc/html_ref_user/index.html`
`$MED_ROOT_DIR/share/salome/doc/MedMemory_user_2on1.pdf`

[3] VTK home page : `http://public.kitware.com/VTK`

[4] Med File Data Model : V. LEFEBVRE, E. FAYOLLE
Définition du modèle d'échange de données MED v2.2
`https://hammi.extra.cea.fr/static/MED/web_med/pdf/NOTE_HI_26_03_012_A.pdf`