GiD Mesh Library

1. GiD Mesh Library documentation . . . 3
1.1 GiD Mesh Library . . . 3
1.1.1 Introduction . . . 3
1.1.2 Library structure . . . 3
1.1.2.1 Handles . . . 4
1.1.2.2 GiDML Modules . . . 4
1.1.2.2.1 Common functions . . . 5
1.1.2.3 Files . . . 6
1.1.3 Programming issues . . . 6
1.1.3.1 Programming language . . . 6
1.1.3.1.1 Linking GiDML with a C\C++ code . . . 6
1.1.3.1.2 Linking GiDML with a Fortran code . . . 6
1.1.3.2 Requirements . . . 7
1.1.3.3 Coding style . . . 7
1.1.3.4 General aspects . . . 7
1.1.3.5 Dependencies . . . 7
1.1.4 How to use the library . . . 7
1.1.4.1 Example of mesh generation module . . . 8
1.1.4.1.1 Variant of the example . . . 10
1.1.5 Terms of use and licencing schema . . . 10
1.2 GiDML IO module . . . 10
1.2.1 Module introduction . . . 11
1.2.2 Synthetic example . . . 11
1.2.3 I/O Structures . . . 11
1.2.3.1 GiDInput and GiDOutput . . . 12
1.2.3.1.1 Common data in GiDInput and GiDOutput . . . 12
1.2.3.2 Callback functions (GiDInput) . . . 14
1.2.3.3 Nodes . . . 14
1.2.3.4 Elements . . . 14
1.2.3.4.1 ELEM_TYPE . . . 15
1.2.3.5 Type of entity . . . 15
1.2.4 API functions . . . 16
1.2.4.1 Module information and data creator . . . 16
1.2.4.1.1 GiDML_IO module information . . . 16
1.2.4.1.2 GiDML module information . . . 17
1.2.4.1.3 Data creator . . . 18
1.2.4.2 Create and delete handles and structures . . . 18
1.2.4.2.1 GiDInput . . . 18
1.2.4.2.2 GiDOutput . . . 21
1.2.4.3 Callback functions . . . 22
1.2.4.4 Mesh definition . . . 25
1.2.4.4.1 Mesh dimension . . . 25
1.2.4.4.2 Mesh nodes . . . 25
1.2.4.4.3 Mesh edges . . . 26
1.2.4.4.4 Mesh faces . . . 28
1.2.4.4.5 Mesh elements . . . 30
1.2.4.4.6 Generic elements . . . 32
1.2.4.5 Additional parameters . . . 34
1.2.4.5.1 Parameters . . . 34
1.2.4.5.2 Attributes . . . 36
1.2.4.6 Write and Read using files . . . 39
1.2.4.7 Other functions . . . 41
1.2.5 Terms of use of the module . . . 42
1.3 Modules . . . 42
1.3.1 Mesh generation . . . 42
1.3.2 Comming soon . . . 42
1.3.2.1 Mesh generation modules . . . 42
1.3.2.2 Mesh editing modules . . . 42
1.3.2.3 Mesh analysis modules . . . 42

GiD Mesh Library documentation

GiD Mesh Library

Introduction

This is the documentation of the **GiD Mesh Library (GiDML)**.
It includes a description of the GiD Mesh Library and its modules, the API description, the coding style and licence schema.

The GiD Mesh Library contains several libraries (GIDML modules) dealing with mesh generation, edition or analysis, to be used by other softwares.

These meshing operations have been conceived in the frame of the numerical simulations, considering several numerical methods and kinds of simulations, leading to several types of meshes: unstructured, cartesian, structured, etc.

Library structure

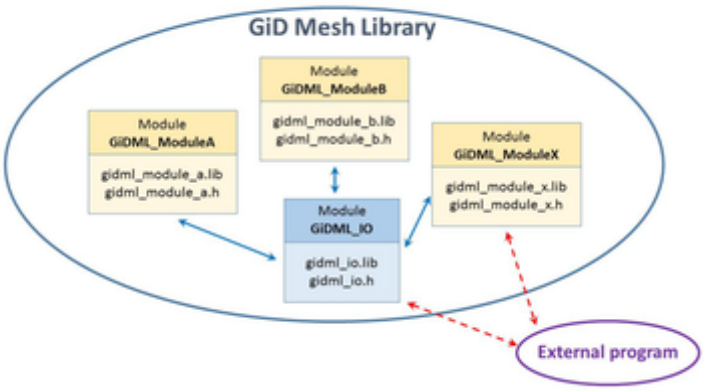
The GiD Mesh Library is formed by a collection of libraries **(modules)** containing several meshing functionalities (for mesh generation, analysis and editing), ready to be called by an external software. (see the list of available modules in GiDML modules).

Each GiDML module consists in a compiled version of the library, and the corresponding header file where all the public functions of the module are declared. The naming convention used for them is the following one:

- gidml_*nameofthemodule*.lib
- gidml_*nameofthemodule*.h

The Input/Output module (**GiDML_IO**) is a special module, as it is mandatory for using any of the other modules of the GiD Mesh Library (see GiDML IO module). All the modules expect to receive the input data in the **GiDInput** format, and return the output data in the **GiDOutput** one. The GiDML_IO module provides the API to use this data structures, allowing any external software to call the functions of any GiDML module. GiDML_IO also offers functions to write and read the data into files (see Write and Read using files), which may be useful to check the data outside the program workflow.

A scheme of the structure of the GiD Mesh Library is depicted hereafter, where the arrows indicate the links between libraries. In this example, the external program is willing to use the module GiDML_ModuleX, so it is linked to it and also to GiDML_IO module.



Structure of the GiD Mesh Library

Handles

All the interaction between the external software and the data for and from the different modules is done using **Handles**. Handles are opaques pointers to GiD Mesh Library internal data structures which are managed by the library.

In the gidml_io.h file (from the GiDML_IO module, see GiDML IO module), the following definitions are made, in order to help the developers to differentiate and deal with the handles:

```
typedef void * GiDInput_Handle;
typedef void * GiDOutput_Handle;
typedef void * GiDIO_Handle;
```

As input and output data structures share several parts, GiDIO_Handle is used in those functions which work on these common parts.

GiDML Modules

Each GiDML module is formed by its library (.lib or .a) and its header public file (.h):

gidml_*nameofthemodule*.lib (MS Windows) or gidml_*nameofthemodule*.a (Linux, macOS)
gidml_*nameofthemodule*.h

As described in Coding style the name of the files are in lower-case, but the name of the module itself can be in Camel-case.
All the public functions of the module are declared in the corresponding header file, using the following naming convention:

GiDML_<NameOfTheModule>_<NameOfTheFunction>

Here an example:

```
const char *GiDML_TetrahedraSmoothing_GetErrorString( const int error_id
);
```

Common functions

The public functions accessible from the module are declared in the gidml_*nameofthemodule*.h file. However, there are some common functions to all the modules. They are presented hereafter.

GetModuleName

This function is used to get at run-time the identifier name of the module.
The declaration of this function is:
const char *GiDML_<*NameOfTheModule*>_GetModuleName();
The function returns a *const char* * with the name of the module which shouldn't be freed nor deleted.

GetModuleVersion

This function is used to get the version of the module.
The declaration of this function is:
const char *GiDML_<*NameOfTheModule*>_GetModuleVersion();
The function returns a const char * with the identifier of the version of the module. The returned string shouldn't be freed nor deleted.
Eventually, a module can provide a function to get their version in a numeric format (int or double) in order to detect easily if the module's version is newer or older than expected.

GetModuleFormatVersion

This function is used to get the version of the format of file used by the module for the input or output data when dumped into a file (see Write and Read using files).
The declaration of this function is:
const char *GiDML_<*NameOfTheModule*>_GetModuleFormatVersion();
The function returns a const char * containing the identifier of the version of file format, which shouldn't be freed nor deleted.

CheckConsistency

This function is used to check the consistency of the input data to be used in the module. The checks done inside this function are light (computationally inexpensive), and only try to detect pathological cases.
For example, if the module to be used generates a tetrahedra volume mesh from a contour mesh made of triangles provided as input, this function would check for a triangle mesh in the input data.
The declaration of this function is:
int GiDML_<*NameOfTheModule*>_CheckConsistency(const GiDInput_Handle hdl_gin);
The function returns 0 if the input data is ok, or an integer different from 0 (i.e. an error_id) if something went wrong in the consistency checking process. To get the error in a human readable string form we can use the function int GiDML_<*NameOfTheModule*>_GetErrorString(int) using the error_id returned from CheckConsistency.

GetErrorString

This function is used to get the error message string corresponding to a specific error_id.
The declaration of this function is:
const char GiDML_<*NameOfTheModule*>_GetErrorString{*}(const int error_id);
The function returns the error message as a const char *. This returned error message shouldn't be freed nor deleted.

Eventually, the corresponding GiDML module may include the collection of messages returned in the public header file, so that they can be managed or used by the end-application, for instance to translate messages, replacement, or others.
A recommended practice is to use error_id equal to 0 for a successful execution of the function, when no problem arises. But it's up to the module to define their error_id's and messages.

DeleteGiDOutputContent

This function is used to delete the content of the output data stored in the GiDOutput_Handle. As this data has been generated by the GiDML module, its deletion must be carried out by the module itself.
The declaration of this function is:
int GiDML_<*NameOfTheModule*>_DeleteGiDOutputContent(GiDOutput_Handle hdl_gout);
The function returns 0 if the content of the GiDOutput_Handle has been deleted succesfully, or an integer different from 0 (i.e. an error_id) otherwise.

Files

In some cases it may be interesting to dump the contents of the *GiDInput* or *GiDOutput* structures into a file, or read it from a file. The *GiDML_IO* module provides some functions to read and write this data from and to files (see Write and Read using files).

The filenames end with the extension .gidml, and are in binary format. Specifically, the HDF5 format is used. They can be examined with any HDF5 viewer software (such as HDFView).

Each version of the *GiDML_IO* module may use a different version of the file-format (the information included or the file structure). The file-format version can be checked with GetModuleFormatVersion and the module version with GetModuleVersion, both included in the section Module information and data creator.

A .gidml file may contain information of a GiDInput, a GiDOutput or both structures.

Programming issues

The GiD Mesh Library bundle provides examples and specific instructions to compile and link external program with the module libraries.

Programming language

The GiD Mesh Library is developed in C++.
The API has been designed to be as simple as possible and to make it easier in its integration in the final application.
The API uses handles (void *) and basic types (double * and int * arrays) to pass the data and point to internal data structures through the different API functions, in order to minimize the programming impact in the final application.

Linking GiDML with a C\C++ code

Include the corresponding gidml_*nameofthemodule*.h header file, and link it with the corresponding gidml_*nameofthemodule*.lib or gidml_*nameofthemodule*.a library file. If the destination application is written in 'C' the C++ libraries (stdc++, stl, etc.) should also be added to the link process.

Linking GiDML with a Fortran code

To use the library in FORTRAN the .h include files are not required, only the library is needed to be added for the linker.

Depending on the version of FORTRAN (e.g. FORTRAN77, FORTRAN90) and the compiler used it could be necessary to write an extra wrapping .F90 interface file, that provide the bindings between FORTRAN and C

(naming conventions and parameters).

Requirements

The requirements and dependencies are specifics to the GiDML module used.
As the GIDML Mesh library is developed in C+, it requires to be linked with the C+ library (stdc++, msvcrt) and eventually the STL (Standard Template Library).

Look for the Requirements page or each module.

in the GIDML_IO's pages:
The GIDML IO module also requires:

- the zlib library version >= 1.2.5.
- the hdf5 and hdf5_hl libraries version >= 1.8.5.

Look into the corresponding COMPILE.txt for more details.

Coding style

All code in the GiDML functions are following the 'Google c++ Style Guide':

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

The naming of the functions follows the rule:

- the first letter of each word of name is capital.

Example: **MyFunction()**;

To identify public visible functions of the library module the prefix '**GiDML_**' is being used. These are the functions which may be called from an external program.

Each input parameter must have the directive **const**, which indicates that it cannot be modified by the function.
The parameter order is: **Input parametrs** followed by **Output parameters**.

Ex: **MyFunction (const p_one, p_two, p_three);**

General aspects

IMPORTANT:All arrays start with zero index.

Dependencies

C++ library and eventually the STL library.
It depends on each GiDML module.
Look for the COMPILE.txt and the corresponding GiDML module page:

- GiDML_IO compilation
- GiDML_OctreeTetrahedraMesher module compilation
- GiDML_Image2Mesh module compilation

How to use the library

The basic steps to use a module **X** of the GiD Mesh Library inside an external program **A** are:

- Include the header files gidml_io.h and gidml_module_x.h in the compilation of **A.**
- Then, **A** can call and use any of the functions declared in the API of gidml_module_x.lib.
- Link **A** with gidml_io.lib and gidml_module_x.lib, and their dependencies (stdc++, stl, mvcrt, etc.).
- gidml_io also depends on third part open source libraries: zlib and hdf5 and hdf5_hl, these libraries are required to link

We will use a simple synthetic example to make it clearer.

Example of mesh generation module

In this example we are suposing that a module for generating a tetrahedra mesh called **GiDML_FakeTetraMesher** is included in the GiD Mesh Library, and we want to use it in our code.
The mesher needs as an input the contour mesh, made of triangles, of the volume to be meshed, and the general element size desired for the volume mesh as a parameter. We will use a hardcoded mesh of triangles for this example, which represents a cube of 10 units of length size.

C++ code of the example

```
#include "gidml_io.h"
#include "gidml_fake_tetra_mesher.h"
#define NAME_OF_MY_PROGRAM "MY_PROGRAM" //identifier of who is creating
the input data
int main() {
  //in this example a cube which contour is defined by 12 faces is used

double coordinates[24]={0,0,0,10,0,0,10,10,0,0,10,0,0,0,10,10,0,10,10,10
,10,0,10,10};

int conectivities[36]={0,1,3,1,3,2,1,2,5,2,5,6,0,4,3,3,4,7,4,5,7,5,6,7,0
,1,4,1,4,5,3,2,7,7,2,6};

  //Create GiDInput Handle.
  const int n_dimension = 3; //space dimension of the data
  const char *module_name = GiDML_FakeTetraMesher_GetModuleName();
  const char *module_format_version =
GiDML_FakeTetraMesher_GetModuleFormatVersion(); //module data
  GiDInput_Handle hdl_input=GiDML_IO_NewGiDInputHandle(module_name,
module_format_version, NAME_OF_MY_PROGRAM, n_dimension);
  //the module name and format are interesting in case that the input is
saved/read to/from a auxiliary file, in order to identify its use

  //Fill GiDInput Handle with data
  const int number_of_nodes = 8;
  const int number_of_faces = 12;
  const ElemType contour_element_type = GID_TRIANGLE_ELEMENT;
  const int nnode_element = 3;
  const double general_size = 2.0;
  GiDML_IO_SetNodesCoords(hdl_input, number_of_nodes,
n_dimension,number_of_nodes,coordinates); //nodes data
```

```
  GiDML_IO_SetFaces(hdl_input, number_of_faces, conectivities,
contour_element_type, nnode_element); //elements data

GiDML_IO_SetParameter(hdl_input,"GENERAL_MESH_SIZE",general_size); //add
a parameter named 'GENERAL_MESH_SIZE'

  //Check the that the Input format is correct.
  int error_returned =
GiDML_FakeTetraMesher_CheckConsistency(hdl_input);

  //GiDOutput Handle.
  GiDOutput_Handle hdl_output = GiDML_IO_NewGiDOutputHandle();
  if ( error_returned == 0){
    //Call the mesher.
    error_returned = GiDML_FakeTetraMesher(hdl_input,hdl_output);
  }

  if ( error_returned )
    const char *error_message =
GiDML_FakeTetraMesher_GetErrorString(error_returned);
    //somehow show this error message...
  } else {
    //Get the tetrahedra mesh
    const int number_of_nodes_in_tetra_mesh = GiDML_IO_GetNumberOfNodes(
hdl_output );
    const int num_of_tetras = GiDML_IO_GetNumberOfElements( hdl_output
);
    double *coords = NULL;
    GiDML_IO_GetNodesCoords( hdl_output,coords );
    int *tetras_connectivity = NULL;
    int fail=GiDML_IO_GetElements( hdl_output,tetras_connectivity);
  }

  //Delete data from GiDOutput
  GiDML_FakeTetraMesher_DeleteGiDOutputContent(hdl_output);

  //Delete data structures inside GiDInput and GiDOutput structures
  GiDML_IO_DeleteGiDInputHandle(hdl_input);
  GiDML_IO_DeleteGiDOutputHandle(hdl_output);
```

```
  return 0;
}
```

Variant of the example

It has to be noted that in the example above, code may be simpler if the function *GiDML_IO_NewGiDInputHandleWithFaces* is used instead of *GiDML_IONewGiDInputHandle*, as it integrates in its parameters the contour mesh.

Using it would replace this code:

```
void* hdl_input=GiDML_IO_NewGiDInputHandle(module_name,module_format_version,NAME_OF_MY_PROGRAM,n_dimension);
GiDML_IO_SetNodesCoords(hdl_input,number_of_nodes,n_dimension,number_of_nodes,coordinates); //nodes data
GiDML_IO_SetFaces(hdl_input,number_of_faces,conectivities,contour_element_type,nnode_element); //elements data
```

by this one

```
void* hdl_input=GiDML_IO_NewGiDInputHandleWithFaces(module_name,module_format_version,NAME_OF_MY_PROGRAM,n_dimension,number_of_nodes,coordinates,number_of_faces,conectivities,contour_element_type,nnode_element);
```

Terms of use and licencing schema

Each GiDML module has its own lincencing schema. It has to be checked in its documentation, and with the owner of each module.

Specifically, the GiDML_IO module (which is property of CIMNE), is free and public, and can be used without any special agreement for any purpose.

GiDML IO module

This is the only mandatory module to use the GiD Mesh Library. It deals with the input and output data for all the other modules, which share the same data structure.

The GiDML_IO module is public and free, and it can be used without restriction, with no need to sign any specific agreement with CIMNE.

The library and the .h file of this module are:

- gidml_io.lib (libgidml_io.a for Linux)
- gidml_io.h

and can be downloaded from ftp://www.gidhome.com/pub/gidml

Note: The source code (C++) of the GiDML_IO module is not public, only the compiled library is provided (for several platforms)

GiDML_IO requires also the following open source third party libraries:

- hdf5 and hdf5_hl libraries
- zlib

These third party libraries can be downloaded from Internet and compiled, but for easy of use we provide a pre-compiled binaries along with the GiDML_IO module.

All the interaction between the external software and the data is done via Handles.

The API functions of the GiDML_IO module allow to fill (Set) and access (Get) the data from the input and output structures.

This document refers to the documentation of the **version 1.10.0 of the GiDML_IO** module.

Module introduction

This is the documentation of the module of the GiDMeshLibrary **GiDML_IO**. It refers to its **version 1.9.0**.

Aditional information can be found at www.gidhome.com/gidml.

For any comment or suggestion, please contact gidml@cimne.upc.edu.

Synthetic example

A synthetic example in c++ of how to use GiDML_IO API functions is shown hereafter:

```
int main() {
  GiDInput_Handle ginput_handle=NULL;
  GiDOutput_Handle goutput_handle=NULL;
  GiDML_IO_NewGiDInput(ginput_handle); //This function returns the
pointer to the GiDInput, the ginput_handle
  MyCode_FillGiDInputWithWhatever(ginput_handle);
  GiDML_IO_NewGiDOutput(goutput_handle); //This function is to get the
hanble of GiDOutput
  GiDML_CalculateSomething(ginput_handle,goutput_handle); //This is a
function of whatever GiDML module, doing 'something' using the data in
ginput_handle, and returning the result in goutput_handle
  MyCode_GetResultFromGiDOutput(goutput); //process the results and save
it as MyCode needs
  MyCode_DeleteWhateverFilledInGiDInput(); //as the data inside GiDInput
is created by MyCode, it is the one who must delete it
  GiDML_IO_DeleteGiDInput(ginput); //This function is deleting the
GiDInput data structure, but not its contents, as it has been created
outside the GiD Mesh Library
  GiDML_CalculateSomething_DeleteGiDOutputContent(goutput);//This
function is deleting the GiDOutput data, as they have been created by
the GiD Mesh Library
  GiDML_IO_DeleteGiDOutput(goutput); //This function is deleting the
GiDOutput data structure
  return 0;
}
```

More detailed examples specific for each GiDML module are presented in Modules.

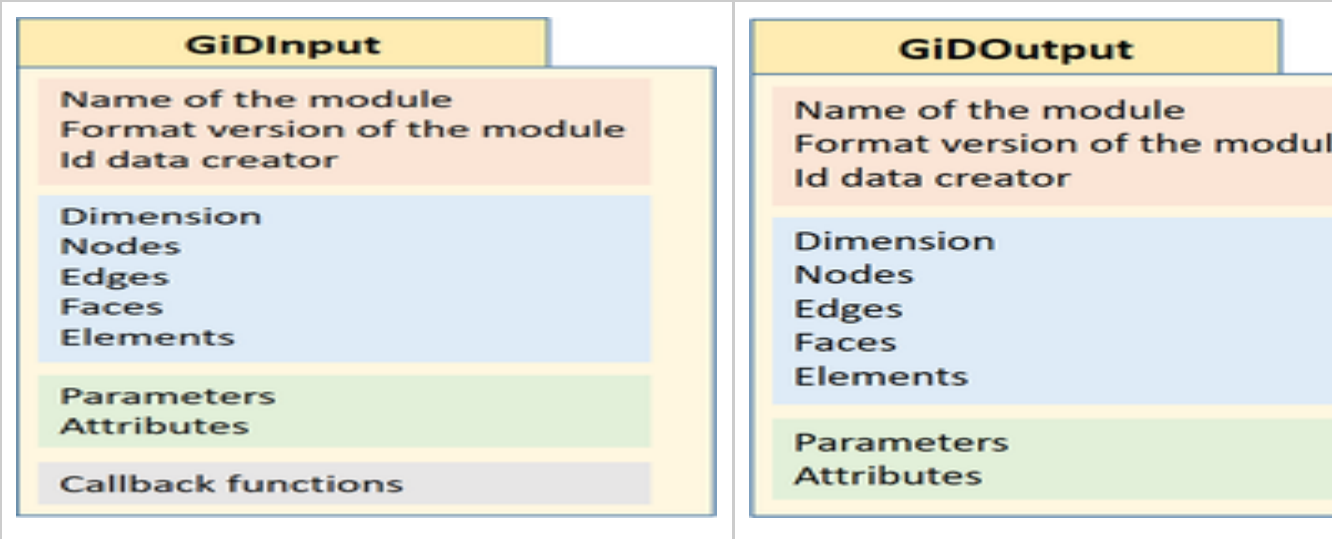
I/O Structures

Hereafter the structures and variables to be used for input and output data are detailed.

For some of the structures and variable names presented hereafter, the word "attributes" appears. The meaning of it is an array (of doubles or integers) attached to mesh entities.

GiDInput and GiDOutput

All the data used in the GiD Mesh Library to communicate with the external solver is grouped into two main structures: GiDInput, and GiDOutput. An overview of it is shown in the following figure:



Not all structures must be filled, usually only some of them are used, the rest are left empty.

Mainly, these two structures have the same structure, although there are some specific data only present in one of them. The data present in GiDInput and GiDOutput structures is detailed hereafter.

Common data in GiDInput and GiDOutput

Module information and data creator

The information to identify the module as well as the creator of the data is presented in this section.

- **Version of the GiDML_IO module and format:** the version of the GiDML_IO module, as well as the version of the format used to dump the data into a file is stored in GiDInput and GiDOutput structures.
- **Name and format version of the module**: it is useful in some situations, to know the gidml_module the data has been created for. The name of the module, as well as the format version are stored. The format version is used when writing the data into, or reading it from a file (see Write and Read using files). The type of variable of the name and the format version is *char.*
- **Identifier of the creator of the data:** an identifier of the creator of the data (in *char* format) is stored also in the GiDInput and GiDOutput data. It is usefull for helping to reproduce some behavior of the corresponding module, or for tracking some data.

Mesh definition

In this section, the data needed to define a mesh is detailed.

- **Spatial dimension:** this refers to the spatial dimension the corresponding module is working with (typically, 2 or 3). It is stored in *int* format.

- **Nodes:** this includes the coordinates of the nodes and more information linked to them (attributes). More details in Nodes.
- **Edges:** this includes the connectivities of the edges (line elements) and more information linked to them (attributes) . More details in Elements.
- **Faces:** this includes the connectivities of the faces (surface elements) and more information linked to them (attributes) . More details in Elements.
- **Elements:** this includes the connectivities of the volume elements and more information linked to them (attributes) . More details in Elements.

Parameters

In this section, the data not directly linked to the mesh entities is detailed:

- **Scalar Parameters:** it is used to include parameters identified by its unique name, needed for the corresponding gidml module, not directly related to the mesh entities. Each gidml module should expect to receive input parameters with the expected name.

scalar parameters are currently stored as a double

For instance: a module can expect in the input data scalar parameters like:

- general mesh size
- a flag indicating if the mesher is constrained or not
- ...

There is an specific constant defined in gidml_io.h file to indicate that a parameter is not valid (or has been not filled). It is #define INVALID_PARAMETER_VALUE DBL_MAX.
Note that there is no need for all the parameters to have a valid value. There can be parameters with no valid ones.

- **Vectorial parameters:** these are arrays of integers or doubles that may be used to include information of array nature, but not related directly to the mesh entities.

An example of this data could be a module expecting to receive in the input data three arrays of coordinates (one for each direction, x, y and z) defining a regular cartesian grid in the 3D space. In this case there should be one vector parameter for each direction

Attributes

Attributes are used in the context of the GiD Mesh Library to refer arrays of integers or doubles with a name that are attached to mesh entities: nodes, edges, faces, or elements. (Data arrays not attached to mesh is called 'Vector parameters' in this library)

It has to be noted that the length of the arrays in implicitly the same as the amount of associated mesh entities. For instance: the ModuleAttributes of the nodes, are always of length equal to the number of nodes.
Although they can be used for different purposes depending on the gidml module, we define two conceptual levels of them:

- **Module attributes:** are the ones expected by the gidml module to be used for some predefined operation.

An example of this could be a module expecting to receive a flag for each node of the input data (it would be a vector of type integer related to the nodes), or the mesh size related to each face of the input (it would be a vector of type double related to the faces).

- **User attributes:** The input data must be somehow transmitted to some output entities created by the module, but usually they are not needed to perform any predefined operation of the module (it they are unknown they doesn't do any operation, are only transferred to the output)

An example of this could be a module mapping nodes onto a collection of triangles (faces). In this case, the input would be a collection of faces and a collection of nodes, and the output the collection of nodes in the final

position (already mapped onto the faces). If that faces has some UserAttribute (integer flag) associated in the GiDInput, the module may return a UserAttribute assigned to the nodes (in the GiDOutput structure), so as the nodes has the flag of the face where they are mapped to.

Callback functions (GiDInput)

Inside GiDInput structure there are also included some pointers to callback functions that can be used to get interactive feed-back with the corresponding gidml module at execution time. These functions are:

```
void SetCallbackStatus(GiDInput_Handle& hdl_gin,
void fn,void client_data);
void GetCallbackStatus(const GiDInput_Handle& hdl_gin,void*& fn,void*&
client_data);
void SetCallbackError(GiDInput_Handle& hdl_gin,void* fn,void*
client_data);
void GetCallbackError(const GiDInput_Handle& hdl_gin,void*& fn,void*&
client_data);
```

A callback function is simply a 'neutral' function, with a predefined prototype of its arguments, that will be called when some event happen.

`SetCallbackStatus` could be used to provide information of the current status of the meshing process, like the percent that is done, the step of the algorithm, its convergence, etc.

`SetCallbackError` could be used to provide information of some error detected by the mesher.

Nodes

The structure for defining the nodes of the input and output meshes contains:

- **Number of nodes:** an integer value indicating the number of nodes (nnodes).
- **Nodes coordinates**: they are in an array of doubles of length equal to ndim * nnodes (where ndim is the spatial dimension, and nnodes the number of nodes).
  Coordinates are interleaved, e.g. in 3D case x0,y0,z0,x1,y1,z1,...
- **Attributes**: a collection of named attributes (arrays of integers or doubles) related with the nodes. Each array has a length equal to nnodes.

Elements

Three categories of elements are used in the input and output data: edges (for line elements), faces (for surface elements) and elements (for volume elements).
The structure for defining the edges, faces and elements of the input and output meshes is the same (Elements). For notation purposes, we will use in this section the term 'elements' when referring to edges, faces and elements, independently of its category.

This structure contains:

- **Number of elements** an integer value indicating the number of elements (nelements).
- **Elements connectivities**: they are in an array of integers referring to the nodes defined in Nodes. The id of each node (the value in the connectivities array) refers to the position of the node in nodes definition. For example, the [2 0 3] connectivity array refers to the third, first and fourth nodes inside Nodes structure (remember the arrays begin by 0, so node indexes too).
- **Number of element types:** the number of different element types included in the structure

(n_elem_types). An element type is considered as a kind of element with the number of nodes it has. It is stored as an integer.

For instance, if we are including triangles of 3 nodes, triangles of 6 nodes and quadrilateral of 4 nodes, we should specify 3 element types.

- **Element types**: this indicates the element types assigned (see ELEM_TYPE for their definitions). It is an array of enums (ElemType) of length equal to n_elem_types.
- **Number of elements by type**: this indicates the number of elements for each element type. It is an array of integers of length equal to n_elem_types.
- **Number of nodes for each element type:** this indicates the number of nodes for each element type. It is an array of integers of length equal to n_elem_types.
- **Attributes**: a collection of attributes (arrays of integers or doubles) related with the elements. Each array of doubles has a length equal to the number of elements.

The elements connectivities corresponds to all the elements sorted so the elements of the same type should be together.
An example is shown for the case there are more than one element type. Imagine we have are 2 triangles of 3 nodes (tri3), 1 triangle of 6 nodes (tri6) and 2 quadrilateral of 4 nodes (qua4), one option will be to put first the tri3, followed by the tri6, and then the qua4 elements as follows:

- Number of elements: 6
- Elements connectivities: { 0 1 2 3 4 5 6 4 0 3 1 5 2 4 8 2 3 5 6 1 8 9 7 }
- Number of element types: 3.
- Element types: {GID_TRIANGLE_ELEMENT GID_TRIANGLE_ELEMENT GID_QUADRILATERAL_ELEMENT}
- Number of elements by type: { 3 1 2 }
- Number of nodes for each element type: { 3 6 4 }

The element connectivities broken down would be like this:
3 triangles tri3 1 triangle tri6 2 quadrilateral quad4
elements connectivitites-> [0 1 2] [3 4 5] [6 4 0] [3 1 5 2 4 8] [2 3 5 6] [1 8 9 7]

ELEM_TYPE

ENUMERATION **ELEM_TYPE**
Enumeration of types. This list have all different type of mesh elements treated by the GiD Mesh Library. These are:
GID_NONE_ELEMENT = 0 //undefined element type
GID_LINE_ELEMENT = 1
GID_TRIANGLE_ELEMENT = 2
GID_QUADRILATERAL_ELEMENT = 3
GID_TETRAHEDRON_ELEMENT = 4
GID_HEXAHEDRON_ELEMENT= 5
GID_PRISM_ELEMENT = 6
GID_POINT_ELEMENT = 7
GID_PYRAMID_ELEMENT = 7
GID_SPHERE_ELEMENT = 8
GID_CIRCLE_ELEMENT = 9

Type of entity

Different types of entities are used in the GiD Mesh Library.

The entity type is defined in gidml_io.h header file, with the following enum:
enum GIDML_TYPE_OF_ENTITY {

GIDML_NODE=0,

GIDML_EDGE=1,

GIDML_FACE=2,

GIDML_ELEMENT=3

};

API functions

As already commented in Handles, the access to the input and output data is done via handles. In the **gidml_io.h** header file, all the needed API functions to set and get these data are provided.
They are detailed hereafter.

Module information and data creator

GiDML_IO module information

These functions refere to the GiDML_IO module itself.

GiDML_IO_GetVersion

Declaration:
const char *GiDML_IO_GetVersion();
Definition:
This function provides with the version of the GiDML_IO module. This version affects to the data of the input and output used in the GiD Mesh Library (don't get confused with the version of a specific GiDML module).
The version of the GiDML_IO module can be seen in the gidml_io.h file, on the #define GiDML_IO_VERSION.
Parameters:
No input nor output parameters used in this function.
The version of GiDML_IO module is returned in a const char* format. It typically is of type: X.Y, where X and Y are integer numbers.

GiDML_IO_GetFormatVersion

Declaration:
const char *GiDML_IO_GetFormatVersion();
Definition:
This function provides with the version of the format of the file used when writing the input and output data into a file (don't get confused with the version of the module GiDML_IO).
The version of the GiDML_IO format can be seen in the gidml_io.h file, on the #define GiDML_IO_FORMAT_VERSION.
Parameters:
No input nor output parameters used in this function.
The version of GiDML_IO format is returned in a const char* format. It typically is of type: X.Y, where X and Y are integer numbers.

GiDML_IO_GetFormatVersion_Number

Declaration:
double GiDML_IO_GetFormatVersion_Number(const char version);
Definition:
This function provides with a number (in double format) corresponding to the version of the GiDML_IO format. It is guaranteed that a newer version will have a number higher than an older one. It is usefull to identify the versioning system with numbers which grow in time, so as checking the number one can identify if one version is older or newer than another.

Parameters:
The input for this function is the version (in const char* format) for which the number is wanted, and returns the number in double format.

GiDML_IO_GetMinimumFormatVersionToSave

Declaration:
const char *GIDML_IO_GetMinimumFormatVersionToSave(const GiDIO_Handle hdl);
Definition:
This function provides with the minimum format version capable to write in file the data included in the input or output data. For instance, if a new version is including an extra information in the nodes structures which is not stored in the corresponding GiDInput structure, the data can be written down with an older format.
Parameters:
The input for this function is the GiDInput or GiDOutput handle (GiDIO_Handle), and it returns the minimum format version that can be used to write down the data inside it (in const char* format).

GiDML module information

These functions refers to the GiDML module the data is used for. For instance: if the data is created as an input for the module *GiDML_FakeTetrahedraMesher (o*bviously, the GiDML_IO module is used to get and set the data for and from the handles), the functions defined hereafter refers to *GiDML_FakeTetrahedraMesher* module (not the GiDML_IO one).

GiDML_IO_GetGiDMeshLibraryModuleName

Declaration:
const char* GiDML_IO_GetGiDMeshLibraryModuleName(const GiDIO_Handle hdl);
Definition:
This function provides with the name of the GiD Mesh Library module the data has been created for (in const char* format).
Parameters:
The input for this function is the GiDInput or GiDOutput handle (GiDIO_Handle), and it returns the GiD Mesh Library module name (in const char* format).

GiDML_IO_GetGiDMeshLibraryModuleFormatVersion

Declaration:
const char* GiDML_IO_GetGiDMeshLibraryModuleFormatVersion(const GiDIO_Handle hdl);
Definition:
This function provides with the version of the format of the file of the corresponding GiDML module used when writing the input and output data into a file (don't get confused with the version of the GiDML module).
Parameters:
The input for this function is the GiDInput or GiDOutput handle (GiDIO_Handle), and it returns the GiD Mesh Library module format version (in const char* format).

GiDML_IO_SetGiDMeshLibraryModuleName

Declaration:
void GiDML_IO_SetGiDMeshLibraryModuleName(const char* module_name,const GiDIO_Handle hdl);
Definition:
This function is used to set the name of the GiD Mesh Library module the data has been created for into the GiDInput or GiDOutput structure.
Parameters:
The input for this function is the module name (in const char* format) and the GiDInput or GiDOutput handle (GiDIO_Handle) of the corresponding GiDInput or GiDOutput structure.

GiDML_IO_SetGiDMeshLibraryModuleFormatVersion

Declaration:
void GiDML_IO_SetGiDMeshLibraryModuleFormatVersion(const char* module_format_version,const GiDIO_Handle hdl);
Definition:
This function set the version of the format of the corresponding GiDML module inside the GiDInput or GiDOutput structure (don't get confused with the version of the GiDML module).
Parameters:
The input for this function is the GiD Mesh Library module format version (in const char* format) 'module_format_version' and the GiDInput or GiDOutput handle (GiDIO_Handle).

Data creator

It is optional for the GiDInput and GiDOutput structures to include an identifier of the creator of the data. This field can be useful when checking files where the data has been dumped to, in order to know the origin the created data.

GiDML_IO_GetGiDMeshLibraryIOCreator

Declaration:
const char GIDML_IO_GetGiDMeshLibraryIOCreator(const GiDIO_Handle hdl, char *io_creator);
Definition:
This function provides with a the identifier of the creator of the data inside the handle (in char* format).
Parameters:
The input for this function is the GiDInput or GiDOutput handle (GiDIO_Handle), and it returns the identifier in the parameter 'io_creator' (in char* format).

GiDML_IO_SetGiDMeshLibraryIOCreator

Declaration:
const char GIDML_IO_SetGiDMeshLibraryIOCreator(const char *gidml_io_creator,const GiDIO_Handle hdl);
Definition:
This function set the identifier of the creator of the data inside the data structure.
Parameters:
The input for this function is the gidml_io_creator (in const char* format) and the GiDInput or GiDOutput handle (GiDIO_Handle) the data creator refers to.

Create and delete handles and structures

GiDInput

//Functions to get and delete GiDInput handle

```
GiDInput_Handle GiDML_IO_NewGiDInputHandle(const GiDInput_Handle
hdl_gin);
GiDInput_Handle GiDML_IO_NewGiDInputHandle(const char *module_name,
const char *module_format_version, const char *io_creator);
GiDInput_Handle GiDML_IO_NewGiDInputHandle(const char *module_name,
const char *module_format_version, const char *io_creator,const int
ndime);
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithMeshNodes(const char
*module_name, const char *module_format_version, const char *io_creator,
const int ndime,const int num_nodes, double *coords);
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithBoundaryFaces(const char
*module_name, const char *module_format_version, const char *io_creator,
const int ndime,const int num_nodes, double *coords, const int
num_faces, int *faces,const ElemType type_faces_mesh, const int nnode);
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithVolumeMesh(const char
*module_name, const char *module_format_version, const char *io_creator,
const int ndime,const int num_nodes, double *coords, const int
num_elements, int *elems,const ElemType mesh_elem_type, const int
nnode);
void GiDML_IO_DeleteGiDInputHandle(GiDInput_Handle &hdl_gin);
//DANGER FUNCTION: ONLY TO BE USED IF THE CONTENT OF GIDINPUT CONTENT IS
CREATED USING 'NEW'
void GiDML_IO_DeleteGiDInputContent(GiDInput_Handle &hdl_gin);
```

GiDML_IO_NewGiDInputHandle

There are three declarations of this function, depending on the parameters provided:
Declaration:

- GiDInput_Handle GiDML_IO_NewGiDInputHandle(const GiDInput_Handle hdl_gin);
- GiDInput_Handle GiDML_IO_NewGiDInputHandle(const char *module_name, const char *module_format_version, const char *io_creator);
- GiDInput_Handle GiDML_IO_NewGiDInputHandle(const char *module_name, const char *module_format_version, const char *io_creator,const int ndime);

Definition:
All these functions create the GiDInput structure, and fill it with the data corresponding to the parameters provided.
Parameters:
All these functions return the GiDInput_Handle of the GiDInput structure created.
The first one receives a GiDInput_Handle, and fill the created GiDInput with a copy of the data from this one. The second one receives as parameters the module name (const char*), module format version (const char*) and the data creator (const char*), and the third one receives also the dimension of the mesh data (const int ndime).

GiDML_IO_NewGiDInputHandleWithNodes
Declaration:
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithNodes(const char *module_name, const char *module_format_version, const char *io_creator, const int ndime,const int num_nodes, double *coords);
Definition:

This function creates the GiDInput structure, and fill it with the nodes coordinates.
Parameters:
The function returns the GiDInput_Handle of the created GiDInput structure, and receives the following parameters as input:

- module_name: this is the name of the GiDML module (const char*)
- module_format_version: this is the format version of the module (const char*)
- io_creator: this is the identifier of the data creator (const char*)
- ndime: the dimension of the mesh (const int)
- num_nodes: the number of nodes (const int)
- coords: the coordinates of the nodes (double*), which is an array of length equal to ndime*num_nodes.

GiDML_IO_NewGiDInputHandleWithFaces

Declaration:
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithFaces(const char *module_name, const char *module_format_version, const char *io_creator, const int ndime,const int num_nodes, double *coords, const int num_faces, int *faces, const ElemType elem_type_faces, const int nnode);
Definition:
This function creates the GiDInput structure, and fill it with the nodes coordinates and faces connectivities.
Parameters:
The function returns the GiDInput_Handle of the created GiDInput structure, and receives the following parameters as input:

- module_name: this is the name of the GiDML module (const char*)
- module_format_version: this is the format version of the module (const char*)
- io_creator: this is the identifier of the data creator (const char*)
- ndime: the dimension of the mesh (const int)
- num_nodes: the number of nodes (const int)
- coords: the coordinates of the nodes (double*), which is an array of length equal to ndime*num_nodes.
- num_faces: the number of faces (const int)
- faces: the connectivities of the faces (int*). This is an array of length equal to num_faces*nnode
- elem_type_faces: the element type of the faces (const ElemType)
- nnode: the number of nodes each face has (const int)

GiDML_IO_NewGiDInputHandleWithVolumeElements

Declaration:
GiDInput_Handle GiDML_IO_NewGiDInputHandleWithVolumeElements(const char *module_name, const char *module_format_version, const char *io_creator, const int ndime,const int num_nodes, double *coords, const int num_elements, int *elems,const ElemType elem_type_elems, const int nnode);
Definition:
This function creates the GiDInput structure, and fill it with the nodes coordinates and volume elements connectivities.
Parameters:
The function returns the GiDInput_Handle of the created GiDInput structure, and receives the following parameters as input:

- module_name: this is the name of the GiDML module (const char*)
- module_format_version: this is the format version of the module (const char*)
- io_creator: this is the identifier of the data creator (const char*)
- ndime: the dimension of the mesh (const int)
- num_nodes: the number of nodes (const int)
- coords: the coordinates of the nodes (double*), which is an array of length equal to ndime*num_nodes.
- num_elements: the number of volume elements (const int)
- elems: the connectivities of the elements (int*). This is an array of length equal to num_elements*nnode
- elem_type_elems: the element type of the volume elements (const ElemType)

- nnode: the number of nodes each volume element has (const int)

GiDML_IO_DeleteGiDInputHandle

Declaration:
void GiDML_IO_DeleteGiDInputHandle(GiDInput_Handle &hdl_gin);
Definition:
This function deletes the GiDInput structure of the handle hdl_gin. In case some of the data inside the GiDInput is created by the GiDMeshLibrary, it is also deleted. This is the case, for instance, of the data created when reading from a file (see Write and Read using files). This is not the case of the data created outside the GiDMeshLibrary and put inside the GiDInput using the API functions of GiDML_IO module. The handle is set to NULL after being deleted.
Parameters:
This function receives a reference to the GiDInput handle (hdl_gin), which is set to NULL after calling the function.

GiDML_IO_DeleteGiDInputContent

Declaration:
void GiDML_IO_DeleteGiDInputContent(GiDInput_Handle &hdl_gin);
Definition:
This function deletes the content of the GiDInput structure of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteGiDInputHandle must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
Parameters:
This function receives as a parameter a reference of the handle (hdl_gin).

GiDOutput

GiDML_IO_NewGiDOutputHandle

There are two declarations of this function, depending on the parameters provided:
Declaration:

- GiDOutput_Handle GiDML_IO_NewGiDOutputHandle(const GiDOutput_Handle hdl_gout);
- GiDOutput_Handle GiDML_IO_NewGiDOutputHandle();

Definition:
All these functions create the GiDOutput structure, and fill it with the data corresponding to the parameters provided.

Parameters:
All these functions return the GiDOutput_Handle of the GiDOutput structure created. In the case of providing with the hdl_gout parameter, the data of the new GiDOutput structure created is copied from it.

GiDML_IO_DeleteGiDOutputHandle

Declaration:
void GiDML_IO_DeleteGiDOutputHandle(GiDOutput_Handle& hdl_gout);
Definition:
This function deletes the GiDOutput structure of the handle hdl_gin. In case some the data inside the GiDOutput is created by the GiDMeshLibrary, it is also deleted. This is the case, for instance, of the data created when reading from a file (see Write and Read using files). The handle is set to NULL after being deleted.
Parameters:
This function receives a reference to the GiDOutput handle (hdl_gout), which is set to NULL after calling the function.

GiDOutput>GiDML_IO_DeleteGiDOutputContent

Declaration:
void GiDML_IO_DeleteGiDOutputContent(GiDOutput_Handle hdl_gout);

Definition:
This function deletes the content of the GiDOutput structure of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteGiDOutputHandle must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
It has to be considered that ,typically, the data inside the GiDOutput structure has been created by a specific GiDML module. In most cases, the corresponding GiDML module will provide in its API with the function GiDML_ModuleName_DeleteOutputContent (see DeleteGiDOutputContent).

Parameters:
This function receives as a parameter a reference of the handle (hdl_gout).

Callback functions

```
//************************************************************************
//Callback functions
//************************************************************************
//callback to be raised to provide the main program information of
current status
//(e.g. to show advance bar and cancel process, or print messages in
console)
void GiDML_IO_SetCallbackStatus(GiDInput_Handle& hdl_gin, void* fn,void*
client_data);
void GiDML_IO_GetCallbackStatus(const GiDInput_Handle& hdl_gin,void*&
fn,void*& client_data);
void GiDML_IO_SetCallbackError(GiDInput_Handle& hdl_gin,void* fn,void*
client_data);
void GiDML_IO_GetCallbackError(const GiDInput_Handle& hdl_gin,void*&
fn,void*& client_data);
```

GiDML_IO_SetCallbackStatus

Declaration:
void GiDML_IO_SetCallbackStatus(GiDInput_Handle& hdl_gin, void *fn,void* client_data);
Definition:
This function sets a function that will be called by the library while the meshing process, to allow provide some feedback to the final user of the current status of the process. The program that uses the library could set them for example to print in console status messages, to show a percent advance bar, etc, and also to allow cancel the meshing from this advance bar
Parameters:

The input parameter for the function is the handle of the GiDInput that will store it, the void* fn pointer to the callback function, and a void* client_data pointer to some extra data to be used by the callback function (that could be NULL if no extra data is needed)

Example:

A Module like GiDML_OctreeTetrahedraMesher could implement the status callback in this way:

```
//define the prototype of the callback function
typedef int (OctreeMesher_Callback_Status) (const double
percentage,const int nnodes, const int nelems,void* client_data);


void OctreeMesher::SetCallback() {
  OctreeMesher_Callback_Status* fn_status=NULL;
  void* client_data_status=NULL;
  GiDML_IO_GetCallbackStatus(hdl_gin,(void*
&)fn_status,client_data_status);
  //store fn_status and client_data_status in variable of the class
  octree_stateproc=fn_status;
  octree_clientdatastate=client_data_status;
  return;
}

int OctreeMesher::ProgressBar() {
   int cancel=0;
   ...
   //invoke the callback
   const int
userstop=octree_stateproc(percentage_done,nnodes,nelems,octree_clientdat
astate);
   if(userstop) return -1;
   ...
   return cancel;
}
```

In the final client of the library GiDML_OctreeTetrahedraMesher implement the callback to show and advance bar

```
//define a function with the same prototype of the previous typedef
int WhileOctreeMesherCallback(const double percentage,const int nnodes,
const int nelems,void* client_data) {
  int
userstop=ShowAdvanceBarAndAllowCancel(percentage,nnodes,nelems,client_da
ta);
  return userstop;
}

void SetMyCallback() {
  //set the status callback function pointing to the client definition

GiDML_IO_SetCallbackStatus(ginput_handle,(void*)WhileOctreeMesherCallbac
k,(void*)&minfo_vol);
}
```

GiDML_IO_GetCallbackStatus

Declaration:
void GiDML_IO_GetCallbackStatus(const GiDInput_Handle& hdl_gin,void*& fn,void*& client_data);
Definition:
This function return in fn the pointer of the current status callback and in client_data the pointer to extra data
Parameters:

The input parameter for the function is the handle of the GiDInput that store the data

The output parametes are the void*& fn pointer to get the callback function, and a void&* client_data pointer to get the extra data to be used by the callback function

GiDML_IO_SetCallbackError

Declaration:
void GiDML_IO_SetCallbackError(GiDInput_Handle& hdl_gin,void* fn,void* client_data);
Definition:
It is similar to GiDML_IO_SetCallbackStatus but to invoke a callback function with an error is detected by the library.

The client program could for example implement it to show some error message
Parameters:

Similar to GiDML_IO_SetCallbackStatus

GiDML_IO_GetCallbackError

Declaration:
void GiDML_IO_GetCallbackError(const GiDInput_Handle& hdl_gin,void*& fn,void*& client_data);
Definition:
It is similar to GiDML_IO_GetCallbackStatus but to handle a callback function for errors instead of status
Parameters:

Similar to GiDML_IO_GetCallbackStatus

Mesh definition

Mesh dimension

GiDML_IO_GetMeshDimension

Declaration:
int GiDML_IO_GetMeshDimension(const GiDIO_Handle hdl);
Definition:
This function returns with the dimension of the mesh present in the GiDInput or GiDOutput structure (typically 2 or 3). It is an integer.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure '*hdl*'.

GiDML_IO_SetMeshDimension

Declaration:
void GiDML_IO_SetMeshDimension(const GiDIO_Handle hdl,const int dimension);
Definition:
This function set the mesh dimension of the data in the GiDInput or GiDOutput of the handle.
Parameters:
The input parameters for the function are the handle of the GiDInput or GiDOutput structure '*hdl*' and the dimension (const int).

Mesh nodes

```
void GiDML_IO_SetNodesCoords(GiDIO_Handle hdl,const int num_nodes,
double *coords);
int GiDML_IO_GetNodesCoords(const GiDIO_Handle hdl_gin, double
*&coords);
int GiDML_IO_GetNumberOfNodes(const GiDIO_Handle hdl);
void GiDML_IO_SetNumberOfNodes(GiDIO_Handle hdl,const int num_nodes);
void GiDML_IO_DeleteNodes(GiDIO_Handle hdl);
//DANGER FUNCTION: ONLY TO BE USED IF THE CONTENT OF GIDOUPUT CONTENT IS
CREATED USING 'NEW'
void GiDML_IO_DeleteNodesContent(GiDIO_Handle *hdl_gio);
```

GiDML_IO_SetNodesCoords

Declaration:
void GiDML_IO_SetNodesCoords(GiDIO_Handle hdl,const int num_nodes, double *coords);
Definition:
This function introduces the array of nodes coordinates into the GiDInput or GiDOutput structure.
Parameters:
The function receives as parameters:

- The handle of the GiDInput or GiDOutput structure (GiDIO_Handle hdl)
- The number of nodes (const int num_nodes)
- The array of coordinates (double *coords). The length of the coordinates array is equal to num_nodes*ndim (where ndim is the dimension of the mesh set using GiDML_IO_SetMeshDimension function defined in Mesh dimension).

GiDML_IO_GetNodesCoords

Declaration:
int GiDML_IO_GetNodesCoords(const GiDIO_Handle hdl, double *&coords);
Definition:
This function provides with the array of coordinates of the nodes.
Parameters:
It receives as input the handle of GiDInput or GiDOutput structure (hdl), and return the coordinates array in a double *.

The function returns 0 (int) if everything is ok.

GiDML_IO_GetNumberOfNodes

Declaration:
int GiDML_IO_GetNumberOfNodes(const GiDIO_Handle hdl);
Definition:
This function provides with the number of nodes in the GiDInput or GiDOutput structure of the handle.
Parameters:
The function receives as input parameter the handle (hdl), and return the number of nodes in an integer.

GiDML_IO_SetNumberOfNodes

Declaration:
void GiDML_IO_SetNumberOfNodes(GiDIO_Handle hdl,const int num_nodes);
Definition:
This function set the number of nodes in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameters are the handle (iDIO_Handle hdl) and the number of nodes (const int num_nodes).

GiDML_IO_DeleteNodes

Declaration:
void GiDML_IO_DeleteNodes(GiDIO_Handle hdl);
Definition:
This function deletes the Nodes structure inside the GiDInput or GiDoutput structure of the handle. In case the data of the nodes (coordinates and possible attributes) has been filled by the GiD Mesh Library, it is also deleted. In case the arrays of data has been created outside the GiD Mesh Library, they are not deleted, and have to be deleted by the one who has created them.
Parameters:
The function receives as parameter the handle (GiDIO_Handle hdl) of the GiDInput or GiDOutput structure.

GiDML_IO_DeleteNodesContent

Declaration:
void GiDML_IO_DeleteNodesContent(GiDIO_Handle *hdl);
Definition:
This function deletes the content of the Nodes structure of GiDInput or GiDOutput of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteNodes must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
Parameters:
This function receives as a parameter a reference of the handle (hdl_gin).

Mesh edges

GiDML_IO_SetEdges

Declaration:
There are two possible options for this function:

- void GiDML_IO_SetEdges(GiDIO_Handle hdl,int num_edges,int* connectivities, const int n_elem_types, ElemType* etypes, int* nnodes, int* n_elems_by_type);
- void GiDML_IO_SetEdges(GiDIO_Handle hdl,const int num_edges,int* connectivities, const ElemType etype, const int nnode);

Definition:
As explained in Elements, there can be different element types in the edges definition. The first declaration of the function is used to include the information of edges into the GiDInput or GiDOutput structures when there are more than one element type, and the second, when there is only one element type (common situation, which reduces the number of input parameters).

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- const int num_edges: the number of edges.
- int* connectivities: the array of connectivities of the edges.
- const int n_elem_types: the number of different element types to be included in the GiDInput or GiDOutput structure.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

For the second definition of the function, only the element type (const ElemType etype) and the number of nodes for element type (const int nnode) is needed (apart from the hdl, num_edges and connectivities).

GiDML_IO_GetEdges

Declaration:

There are two possible options for this function:

- int GiDML_IO_GetEdges(const GiDIO_Handle hdl, int*& connec, ElemType*& etypes, int*& nnodes, int*& n_elems_by_type);
- int GiDML_IO_GetEdges(const GiDIO_Handle hdl, int*& connec);

Definition:
As explained in Elements, there can be different element types in the elements definition. The first declaration of the function is used to get the complete information about the elements present in the GiDInput or GiDOutput structure of the handle. In the case of knowing that there is only one element type, the second function can be used for sake of simplicity.

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- int* connectivities: the array of connectivities of the edges.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.

- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

GiDML_IO_GetNumberOfEdges

Declaration:
int GiDML_IO_GetNumberOfEdges(const GiDIO_Handle hdl);
Definition:
This function returns the number of edges present in the GiDInput or GiDOutput of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_GetNumberOfElementTypesInEdges

Declaration:
int GiDML_IO_GetNumberOfElementTypesInEdges(const GiDIO_Handle hdl);
Definition:
This function provides with the number of different element types of the edges present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_DeleteEdges

Declaration:
void GiDML_IO_DeleteEdges(GiDIO_Handle hdl);
Definition:
This function deletes the Edges structure inside the GiDInput or GiDoutput structure of the handle. In case the data of the edges (connectivities, element type information and possible attributes or markers) has been filled by the GiD Mesh Library, it is also deleted. In case the arrays of data has been created outside the GiD Mesh Library, they are not deleted, and have to be deleted by the one who has created them.
Parameters:
The function receives as parameter the handle (GiDIO_Handle hdl) of the GiDInput or GiDOutput structure.

GiDML_IO_DeleteEdgesContent

Declaration:
void GiDML_IO_DeleteEdgesContent(GiDIO_Handle& hdl);
Definition:
This function deletes the content of the Edges structure of GiDInput or GiDOutput of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteEdges must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
Parameters:
This function receives as a parameter a reference of the handle (hdl_gin).

Mesh faces

GiDML_IO_SetFaces

Declaration:
There are two possible options for this function:

- void GiDML_IO_SetFaces(GiDIO_Handle hdl,int num_faces,int* connectivities, const int n_elem_types, ElemType* etypes, int* nnodes, int* n_elems_by_type);
- void GiDML_IO_SetFaces(GiDIO_Handle hdl,const int num_faces,int* connectivities, const ElemType etype, const int nnode);

Definition:
As explained in Elements, there can be different element types in the faces definition. The first declaration of the function is used to include the information of faces into the GiDInput or GiDOutput structures when there are more than one element type, and the second, when there is only one element type (common situation, which reduces the number of input parameters).

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- const int num_faces: the number of faces.
- int* connectivities: the array of connectivities of the faces.
- const int n_elem_types: the number of different element types to be included in the GiDInput or GiDOutput structure.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

For the second definition of the function, only the element type (const ElemType etype) and the number of nodes for element type (const int nnode) is needed (apart from the hdl, num_faces and connectivities).

GiDML_IO_GetFaces

Declaration:

There are two possible options for this function:

- int GiDML_IO_GetFaces(const GiDIO_Handle hdl, int*& connec, ElemType*& etypes, int*& nnodes, int*& n_elems_by_type);
- int GiDML_IO_GetFaces(const GiDIO_Handle hdl, int*& connec);

Definition:
As explained in Elements, there can be different element types in the elements definition. The first declaration of the function is used to get the complete information about the elements present in the GiDInput or GiDOutput structure of the handle. In the case of knowing that there is only one element type, the second function can be used for sake of simplicity.

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- int* connectivities: the array of connectivities of the faces.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It

is an array of length equal to n_elem_types.

GiDML_IO_GetNumberOfFaces

Declaration:
int GiDML_IO_GetNumberOfFaces(const GiDIO_Handle hdl);
Definition:
This function returns the number of faces present in the GiDInput or GiDOutput of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_GetNumberOfElementTypesInFaces

Declaration:
int GiDML_IO_GetNumberOfElementTypesInFaces(const GiDIO_Handle hdl);
Definition:
This function provides with the number of different element types of the faces present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_DeleteFaces

Declaration:
void GiDML_IO_DeleteFaces(GiDIO_Handle hdl);
Definition:
This function deletes the Faces structure inside the GiDInput or GiDoutput structure of the handle. In case the data of the faces (connectivities, element type information and possible attributes or markers) has been filled by the GiD Mesh Library, it is also deleted. In case the arrays of data has been created outside the GiD Mesh Library, they are not deleted, and have to be deleted by the one who has created them.
Parameters:
The function receives as parameter the handle (GiDIO_Handle hdl) of the GiDInput or GiDOutput structure.

GiDML_IO_DeleteFacesContent

Declaration:
void GiDML_IO_DeleteFacesContent(GiDIO_Handle& hdl);
Definition:
This function deletes the content of the Faces structure of GiDInput or GiDOutput of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteFaces must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
Parameters:
This function receives as a parameter a reference of the handle (hdl_gin).

Mesh elements

GiDML_IO_SetElements

Declaration:
There are two possible options for this function:

- void GiDML_IO_SetElements(GiDIO_Handle hdl,int num_elems,int* connectivities, const int n_elem_types, ElemType* etypes, int* nnodes, int* n_elems_by_type);

- void GiDML_IO_SetElements(GiDIO_Handle hdl,const int num_elems,int* connectivities, const ElemType etype, const int nnode);

Definition:
As explained in Elements, there can be different element types in the elements definition. The first declaration of the function is used to include the information of elements into the GiDInput or GiDOutput structures when there are more than one element type, and the second, when there is only one element type (common situation, which reduces the number of input parameters).

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- const int num_elems: the number of elements.
- int* connectivities: the array of connectivities of the elements.
- const int n_elem_types: the number of different element types to be included in the GiDInput or GiDOutput structure.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

For the second definition of the function, only the element type (const ElemType etype) and the number of nodes for element type (const int nnode) is needed (apart from the hdl, num_elems and connectivities).

GiDML_IO_GetElements

Declaration:
There are two possible options for this function:

- int GiDML_IO_GetElements(const GiDIO_Handle hdl, int*& connec, ElemType*& etypes, int*& nnodes, int*& n_elems_by_type);
- int GiDML_IO_GetElements(const GiDIO_Handle hdl, int*& connec);

Definition:
As explained in Elements, there can be different element types in the elements definition. The first declaration of the function is used to get the complete information about the elements present in the GiDInput or GiDOutput structure of the handle. In the case of knowing that there is only one element type, the second function can be used for sake of simplicity.

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- int* connectivities: the array of connectivities of the elements.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

GiDML_IO_GetNumberOfElements

Declaration:

int GiDML_IO_GetNumberOfElements(const GiDIO_Handle hdl);
Definition:
This function returns the number of elements present in the GiDInput or GiDOutput of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_GetNumberOfElementTypesInElements

Declaration:
int GiDML_IO_GetNumberOfElementTypesInElements(const GiDIO_Handle hdl);
Definition:
This function provides with the number of different element types of the elements present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameter for the function is the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl).

GiDML_IO_DeleteElements

Declaration:
void GiDML_IO_DeleteElements(GiDIO_Handle hdl);
Definition:
This function deletes the Elements structure inside the GiDInput or GiDoutput structure of the handle. In case the data of the elements (connectivities, element type information and possible attributes or markers) has been filled by the GiD Mesh Library, it is also deleted. In case the arrays of data has been created outside the GiD Mesh Library, they are not deleted, and have to be deleted by the one who has created them.
Parameters:
The function receives as parameter the handle (GiDIO_Handle hdl) of the GiDInput or GiDOutput structure.

GiDML_IO_DeleteElementsContent

Declaration:
void GiDML_IO_DeleteElementsContent(GiDIO_Handle& hdl);
Definition:
This function deletes the content of the Elements structure of GiDInput or GiDOutput of the handle. It has to be considered that only the content (the arrays of int's and double's) are deleted, but not the data structure. For deleting it the function GiDML_IO_DeleteElements must be called.
NOTE: the content data (which are arrays) will be deleted using the c++ delete[] operator. In case the content has been created outside the GiD Mesh Library, make sure that it has been created with new operator.
Parameters:
This function receives as a parameter a reference of the handle (hdl_gin).

Generic elements

As explained in Elements, edges, faces and elements share the same structure. Some API functions are present in this section to be called with 'generic elements' (edges, faces or elements), just indicating the type of them in an input parameter (see Type of entity).

GiDML_IO_SetGenericElements

There are two possible options for this function:

- void GiDML_IO_SetGenericElements(GiDIO_Handle hdl,const GIDML_TYPE_OF_ENTITY type,int num_elems, int* connectivities, const int n_elem_types, ElemType* etypes, int* nnodes, int* n_elems_by_type);

- void GiDML_IO_SetGenericElements(GiDIO_Handle hdl,const GIDML_TYPE_OF_ENTITY type, const int num_elems,int* connectivities, const ElemType etype, const int nnodes);

Definition:
As explained in Elements, there can be different element types in the generic elements definition. The first declaration of the function is used to include the information of generic elements into the GiDInput or GiDOutput structures when there are more than one element type, and the second, when there is only one element type (common situation, which reduces the number of input parameters).
Of course, one can also use the first declaration when only one element type is considered. The second declaration is only for simplification purposes.

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- const GIDML_TYPE_OF_ENTITY type: the type of generic element considered.
- const int num_elems: the number of generic elements.
- int* connectivities: the array of connectivities of the generic elements.
- const int n_elem_types: the number of different generic element types to be included in the GiDInput or GiDOutput structure.
- ElemType* etypes: the array of the different generic element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per generic element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of generic elements for each element type to be included. It is an array of length equal to n_elem_types.

For the second definition of the function, only the element type (const ElemType etype) and the number of nodes for element type (const int nnode) is needed (apart from the hdl, num_elems and connectivities).

GiDML_IO_GetGenericElements

Declaration:

There are two possible options for this function:

- int GiDML_IO_GetGenericElements(const GiDIO_Handle hdl, const GIDML_TYPE_OF_ENTITY type, int*& connec, ElemType*& etypes, int*& nnodes, int*& n_elems_by_type);
- int GiDML_IO_GetGenericElements(const GiDIO_Handle hdl, const GIDML_TYPE_OF_ENTITY type, int*& connec);

Definition:
As explained in Elements, there can be different element types in the elements definition. The first declaration of the function is used to get the complete information about the elements present in the GiDInput or GiDOutput structure of the handle. In the case of knowing that there is only one element type, the second function can be used for sake of simplicity.

Parameters:

- GiDIO_Handle hdl: handle of the GiDInput or GiDOutput structure.
- const GIDML_TYPE_OF_ENTITY type: the type of generic element considered.
- int* connectivities: the array of connectivities of the elements.
- ElemType* etypes: the array of the different element types to be included. It is an array of length equal to n_elem_types.
- int* nnodes: the array defining the number of nodes per element for the different element types included. It is an array of length equal to n_elem_types.
- int* n_elems_by_type: the array defining the number of elements for each element type to be included. It is an array of length equal to n_elem_types.

GiDML_IO_GetNumberOfGenericElements

Declaration:
int GiDML_IO_GetNumberOfGenericElements(const GiDIO_Handle hdl, const GIDML_TYPE_OF_ENTITY type);
Definition:
This function returns the number of generic elements (edges, faces or elements) present in the GiDInput or GiDOutput of the handle.
Parameters:
The input parameters for the function are the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl) and the type of element considered (const GIDML_TYPE_OF_ENTITY type)

GiDML_IO_GetNumberOfGenericElementTypes

Declaration:
int GiDML_IO_GetNumberOfGenericElementTypes(const GiDIO_Handle hdl, const GIDML_TYPE_OF_ENTITY type);
Definition:
This function provides with the number of different generic element types present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameters for the function are the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl) and the type of element considered (const GIDML_TYPE_OF_ENTITY type).

GiDML_IO_GetGenericElementType

Declaration:
ElemType GiDML_IO_GetGenericElementType(const GiDIO_Handle hdl,const GIDML_TYPE_OF_ENTITY type, const int ipos);
Definition:
This function provides with a specific element type present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameters for the function are the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl), the type of element considered (const GIDML_TYPE_OF_ENTITY type) and the position of the element type consulted inside the etypes array (const int ipos). Note that ipos must be an integer between 0 (0 included) and the number of generic element types (not included). This is: ipos>=0 and ipos<n_elem_types.

GiDML_IO_GetNNodeGenericElement

Declaration:
int GiDML_IO_GetNNodeGenericElement(const GiDIO_Handle hdl,const GIDML_TYPE_OF_ENTITY type, const int ipos);
Definition:
This function provides with the number of nodes for a specific element type present in the GiDInput or GiDOutput structure of the handle.
Parameters:
The input parameters for the function are the handle of the GiDInput or GiDOutput structure (const GiDIO_Handle hdl), the type of element considered (const GIDML_TYPE_OF_ENTITY type) and the position of the element type consulted inside the etypes array (const int ipos). Note that ipos must be an integer between 0 (0 included) and the number of generic element types (not included). This is: ipos>=0 and ipos<n_elem_types.

Additional parameters

Parameters

GiDML_IO_SetParameter / GiDML_IO_SetParameterVector

Declaration:

These functions can be used to define named global parameters which can take scalar o vector values. Below is listed the function prototypes:

- `int GiDML_IO_SetParameter(GiDIO_Handle hdl, const char* parameter_name, double value);`
- `int GiDML_IO_SetParameter(GiDIO_Handle hdl, const char* parameter_name, int value);`
- `int GiDML_IO_SetParameterVector(GiDIO_Handle hdl,const char* parameter_name,double* array_of_values,const int array_dimension);`
- `int GiDML_IO_SetParameterVector(GiDIO_Handle hdl,const char* parameter_name,int* array_of_values,const int array_dimension);`

Definition:

These functions set a specific parameter with a unique name in the GiDInput or GiDOutput structure of the given handle. The functions with the suffix "Vector" allow to set a vector value. The scalar values are stored always as double.

Parameters:

- GiDIO_Handle hdl: it is the handle of the GiDInput or GiDOutput structure
- const char* parameter_name: it is an unique name to access to this parameter
- double/int value: it is the value of the parameter.
- in case of parameter vectors, also the dimension of the array (int array_dimension) is added.

GiDML_IO_GetParameter / GiDML_IO_GetParameterVector

Declaration:

- int GiDML_IO_GetParameter(const GiDIO_Handle hdl, const char* parameter_name, double& value);
- int GiDML_IO_GetParameterVector(const GiDIO_Handle hdl,const char* parameter_name,double*& array_of_values,int& array_dimension);
- int GiDML_IO_GetParameterVector(const GiDIO_Handle hdl, const char* parameter_name,int*& array_of_values, int& array_dimension)

Definition:

This function provides with the value of the parameter given its name in the parameters array of the GiDInput or GiDOutput structure of the handle.

Parameters:

The function receives as input the following parameters:

- GiDIO_Handle hdl: it is the handle of the GiDInput or GiDOutput structure
- const char* parameter_name: it is an unique name to access to this parameter

The value of the parameter or the array is returned in the argument 'value' or 'array_of_values' (which is a reference), also the array dimension is returned in the argument int& array_dimension

The function returns 0 if the parameter exists, and 1 if it does not exist.

GiDML_IO_GetNumberOfParameters

Declaration:
int GiDML_IO_GetNumberOfParameters(const GiDIO_Handle hdl);
Definition:

This function provides with the amount of parameters in GiDInput or GiDOutput structure.

Parameters:

This function receives as an input the handle (const GiDIO_Handle hdl), and returns the length of the array in integer format.

GiDML_IO_ParameterExists

Declaration:
bool GiDML_IO_ParameterExists(const GiDIO_Handle hdl, const char* parameter_name);
Definition:

This function indicates if a named parameter of the GiDInput or GiDOutput structure has been defined.
Parameters:

The function receives as input the following parameters:

- GiDIO_Handle hdl: it is the handle of the GiDInput or GiDOutput structure
- const char* parameter_name: it is an unique name to access to this parameter

The function returns 'true' if the parameter exists (has been set), or 'false' if not.

GiDML_IO_DeleteParametersContent

Declaration:
void GiDML_IO_DeleteParametersContent(GiDIO_Handle hdl);
Definition:
This function release the memory resources used for all parameters of the GiDInput and GiDOutput structure of the handle.
Parameters:

This function receives as an input the handle (const GiDIO_Handle hdl).

Attributes

As explained in I/O Structures, there may be attributes assigned to Nodes, Edges, Faces and Elements. In this section we describe the functions to deal with the attributes.

An attribute is an array of values which can defined for any of the mesh entities types, either nodes, edges, faces or elements. The array contains one element for each mesh entity, and this determine the size of the array, for example if the attribute array is defined on nodes then the size of the array must be equal to the number of nodes in the mesh. Currently only integer or double values can be stored in any attribute array.

Besides the type of the array element we must specify the type of the array. There are two possible types:

- **User attributes** are transmitted from the input to the output in a 'neutral way', the content of this kind of array does not affect the meshing algorithm specific to the module.
- **Module attributes** are not transmitted to the output, and usually have a meaning for the module to do some action (e.g. input required mesh sizes)

GiDML_IO_CreateAttribute

Declaration:

```
void* GiDML_IO_CreateAttribute(GiDIO_Handle hdl,const char* attribute_name, void*
attribute_array,
const GIDML_TYPE_OF_ENTITY entity_type,
const GIDML_TYPE_OF_ATTRIBUTE attribute_type,
const GIDML_TYPE_OF_VALUE value_type);
```

Definition:

This function creates an attribute_array corresponding to the unique name 'attribute_name'. The elements are initialized to 0.

Parameters:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- const char* attribute_name: The unique name to access to the attribute
- GIDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to (GIDML_NODE, GIDML_EDGE, GIDML_FACE, GIDML_ELEMENT)
- GIDML_TYPE_OF_ATTRIBUTE attribute_type: the category of attribute (GIDML_USER_ATTRIBUTE or GIDML_MODULE_ATTRIBUTE)
- GIDML_TYPE_OF_VALUE value_type: the type of the array data (GIDML_TYPE_INTEGER, GIDML_TYPE_DOUBLE)

The function returns a pointer to the attribute array.

GiDML_IO_SetAttribute

Declaration:

```
int GiDML_IO_SetAttribute(GiDIO_Handle hdl,const char* attribute_name, void*
attribute_array,
const GIDML_TYPE_OF_ENTITY entity_type,
const GIDML_TYPE_OF_ATTRIBUTE attribute_type,
const GIDML_TYPE_OF_VALUE value_type);
```

Definition:

This function sets a given attribute array with an unique name. The attribute array is copied inside the GiDIO structure. The data buffer should have been allocated and initialized previously.

Parameters:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- const char* attribute_name: The unique name to access to the attribute
- void* attribute_array: a pointer to the array values (double* or integer*). Its dimension is the number of entities present in the corresponding entity type structure. For instance: if type=GIDML_NODE, the dimension of this array is equal to the number of nodes.
- GIDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to (GIDML_NODE, GIDML_EDGE, GIDML_FACE, GIDML_ELEMENT)
- GIDML_TYPE_OF_ATTRIBUTE attribute_type: the category of attribute (GIDML_USER_ATTRIBUTE or GIDML_MODULE_ATTRIBUTE)
- GIDML_TYPE_OF_VALUE value_type: the type of the array data (GIDML_TYPE_INTEGER, GIDML_TYPE_DOUBLE)

GiDML_IO_GetAttribute

Declaration:

```
void* GiDML_IO_GetAttribute(GiDIO_Handle hdl,const char* attribute_name,const
GIDML_TYPE_OF_ENTITY entity_type);
```

Definition:
This function returns the buffer data for the given name of the attribute array.

Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- const char* attribute_name: The name of the attribute array
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.

Returns the array of user attributes as void* if must be casted to int or double before using it. You can find the type of the elements by calling the function GiDML_IO_GetAttributeType. NULL is returned in case the attribute array does not exists.

GiDML_IO_GetAttributeType

Declaration:

```
GIDML_TYPE_OF_VALUE GiDML_IO_GetAttributeType(GiDIO_Handle hdl,const char*
attribute_name,const GIDML_TYPE_OF_ENTITY entity_type);
```

Definition:
This function returns the type of the attribute (integer or double) given the name of the attribute array.

Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- const char* attribute_name: The name of the attribute array
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.

returns the type of the array element which could be GIDML_TYPE_INTEGER or GIDML_TYPE_DOUBLE. If the attribute array does not exists it return GIDML_TYPE_UNKNOWN.

GiDML_IO_DeleteAttribute

Declaration:

```
int GiDML_IO_DeleteAttribute(GiDIO_Handle hdl,const char* attribute_name,const
GIDML_TYPE_OF_ENTITY entity_type);
```

Definition:

This function release the memory used by the attribute with the specified name
Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- const char* attribute_name: The unique name to access to the attribute
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.

GiDML_IO_DeleteAttributesEntitiesVector

Declaration:

```
int GiDML_IO_DeleteAttributesEntitiesVector(GiDIO_Handle hdl,const
GIDML_TYPE_OF_ENTITY entity_type);
```

Definition:

This function release the memory used by all attributes defined for the given entity type.

Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.

GiDML_IO_GetNumberOfAttributesThatMustBeTransmitted

Declaration:

```
int GiDML_IO_GetNumberOfAttributesThatMustBeTransmitted(GiDIO_Handle hdl,const
GIDML_TYPE_OF_ENTITY entity_type);
```

Definition:

This function returns the number of user attributes defined (the ones that will be transmitted to the output).

Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.

Returns the number of user attributes.

GiDML_IO_GetAttributeNameThatMustBeTransmitted

Declaration:

```
const char* GiDML_IO_GetAttributeNameThatMustBeTransmitted(GiDIO_Handle hdl,const
GIDML_TYPE_OF_ENTITY entity_type,const int i);
```

Definition:

This function returns the name of the user attribute (the ones that must be transmitted to the output) from its integer index. This index must be in the range [0,n), where n is the number os user attributes defined.

Parameters:

The function receives as input:

- GiDIO_Handle hdl: The handle of the GiDInput or GiDOutput structure
- GiDML_TYPE_OF_ENTITY entity_type: the type of entity the user attributes array refers to.
- int i: the integer id, from 0 to GiDML_IO_GetNumberOfAttributesThatMustBeTransmitted

returns the attribute name as string.

Write and Read using files

GiDML_IO_WriteGiDInput

Declaration:

```
int GiDML_IO_WriteGiDInput(const char* filename,const GiDInput_Handle hdl);
```

Definition:
This function writes the content of a GiDInput structure into a .gidml binary file (see Files).

Parameters:
The function receives as input parameters the name of the file to be created and the handle for the GiDInput structure. The file name must include the extension.
The function returns 1 in case of error, otherwise it returns 0.

GiDML_IO_ReadGiDInput

Declaration:

```
int GiDML_IO_ReadGiDInput(const char* filename, GiDInput_Handle& hdl_gin);
```

Definition:
This function reads the content of a .gidml file (see Files) and fill the GiDInput structure of the handle with this data.

Parameters:
The function receives as input parameters the name of the file to be read and a reference to the handle of the GiDInput structure ('hgl_gin').
Note: hdl_gin must be NULL when calling the function. Inside the function it is created.

The function returns 1 in case of error, otherwise it returns 0.

GiDML_IO_WriteGiDOutput

Declaration:

```
int GiDML_IO_WriteGiDOutput(const char* filename, const GiDOutput_Handle hdl_gout);
```

Definition:
This function writes the content of a GiDOutput structure into a .gidml binary file (see Files).

Parameters:

The function receives as input parameters the name of the file to be created and the handle for the GiDOutput structure. The file name must include the extension.

The function returns 1 in case of error, otherwise it returns 0.

GiDML_IO_ReadGiDOutput

Declaration:

```
int GiDML_IO_ReadGiDOutput(const char* filename, GiDOutput_Handle& hdl_gout);
```

Definition:

This function reads the content a .gidml file (see Files) and fill the GiDOutput structure of the handle with this data.

Parameters:

The function receives as input parameters the name of the file to be read and a reference to the handle of the GiDOutput structure ('hgl_gout').
Note: hdl_gout must be NULL when calling the function. Inside the function it is created.
The function returns 1 in case of error, otherwise it returns 0.

GiDML_IO_WriteGiDInputAndGiDOutput

Declaration:

```
int GiDML_IO_WriteGiDInputAndGiDOutput(const char* filename,const GiDInput_Handle
hdl_gin, const GiDOutput_Handle hdl_gout);
```

Definition:

This function writes the content of the GiDInput and GiDOutput structures of the handles in a .gidml binary file (see Files).

Parameters:
The function receives as input parameters the name of the file to be created and filled with the data of the GiDInput and GiDOutput structure and the handles for both structures
The function returns 1 in case of error, otherwise it returns 0.

GiDML_IO_ReadGiDInputAndGiDOutput

Declaration:

```
int GiDML_IO_ReadGiDInputAndGiDOutput(const char* filename, GiDInput_Handle&
hdl_gin, GiDOutput_Handle& hdl_gout);
```

Definition:

This function reads the content a .gidml file (see Files) and fill the GiDOInput and GiDOutput structures with this data.

Parameters:

The function receives as input parameters the name of the file to be read and a reference to the handle for both GiDInput and GiDOutput structures.
Note: hdl_gin and hdl_gout must be NULL when calling the function. Inside the function they are created.
The function returns 1 in case of error, otherwise it returns 0.
If the file does not contain GiDInput data or GiDOutput data, the corresponding handle remains as NULL.

GiDML_IO_WriteMeshInGiDFormat

Declaration:

```
int GiDML_IO_WriteMeshInGiDFormat(const char* filename, const GiDIO_Handle hdl);
```

Definition:
This function writes the content of the GiDInput or GiDOutput structure of the handle in an ASCII file following the GiD mesh format (.msh). This format is documented in the GiD manuals, as well as in the website www.gidhome.com/support. This may be useful to visualize a mesh from a GiDInput or GiDOutput structure, but it has to be considered that in these structures there should be more information than the needed to write the mesh in GiD format, so additional parameters and other information may not be present in the mesh file.

Parameters:
The function receives as input parameters the name of the file to be created and filled with the mesh coming from the data of the GiDInput structure and the handle of it.
The function returns 1 in case of error, otherwise it returns 0.

Other functions

GiDML_IO_GiDInputsAreEqual

Declaration:

```
bool GiDML_IO_GiDInputsAreEqual(const GiDInput_Handle ginput_handle_1,const
GiDInput_Handle ginput_handle_2);
```

Definition:
This function checks if the data inside two handles of GiDInput structure are equivalent.

Parameters:
It receives the two handles of the GiDInput structures to compare ('ginput_handle_1' and 'ginput_handle_2'), and return a boolean (true or false) indicating if the data inside the structures are equivalent or not.

GiDML_IO_GiDOutputsAreEqual

Declaration:

```
bool GiDML_IO_GiDOutputsAreEqual(const GiDOutput_Handle goutput_handle_1,const
GiDOutput_Handle goutput_handle_2);
```

Definition:
This function checks if the data inside two handles of GiDOutput structure are equivalent.

Parameters:
It receives the two handles of the GiDOutput structures to compare ('goutput_handle_1' and 'goutput_handle_2'), and return a boolean (true or false) indicating if the data inside the structures are equivalent or not.

Terms of use of the module

CIMNE is the propietary of this GiDML module. It is **public and free**, and it can be used without restrictions for any purpose.

Modules

Hereafter, the functions present in the GiD Mesh Library are documented

Mesh generation

List of modules available for mesh generation (documentation accessible from the links):

- GiDML_OctreeTetrahedraMesher module Home
- GiDML_Image2Mesh module Home

Comming soon

Mesh generation modules

List of modules comming soon for mesh generation(documentation accessible from the links):

- GiDML_BoundaryLayerMesher Home
- GiDML_AdvancingFrontTetrahedraMesher Home

Mesh editing modules

List of modules comming soon for mesh editing (documentation accessible from the links):

- GiDML_ProjectNodes module Home

Mesh analysis modules

List of modules comming soon for mesh analysis (documentation accessible from the links):

- GiDML_NodesDistancesToSurfaceMesh Home