An easy way to access files
in Gamma Mesh Format

The libMeshb library

Loïc MARÉCHAL / INRIA, Gamma Project

December 2016

Document v1.6

Library v7.2

Contents

Cover picture: A variety of magnetic tape drives from the 1970's including the famous DECtapes.

1 Introduction

1.1 What is the Gamma Mesh Format ?

The Gamma Mesh Format (GMF) and the associated library *libMeshb* provide programers of simulation and meshing softwares with an easy way to store their meshes and physical solutions.

The GMF features more than 50 kinds of data types, ranging from vertex to polyhedron or normal vectors to matricial solution fields.

The *libMeshb* provides a convenient way to move data between those files, via keyword tags, and the user's own structures.

1.2 An evolutive keyword based format

The GMF is a keyword based file format, meaning that a mesh file consists of a list of keywords, each followed by its data. No keyword is mandatory and a file may contain any combination of them. Furthermore, new keywords may be added while keeping upward and backward compatibility.

It means that older files can be accessed by newer version of the library and vice versa.

1.3 A comprehensive C library

The *libMeshb* provides programmers with a comprehensive set of commands and keywords covering most common operations on many different kinds of mesh or physical solution related data.

Reading, writing and querying files is easily done by calling a couple of commands which are provided in an ANSI C file "libmeshb7.c" and a header file "libmeshb7.h". All is needed is compiling those files along with the calling software.

Fortran APIs are also available: "libmeshb7.ins" for F77 and F90.

1.4 ASCII vs. Binary

GMF files can be stored in ascii or binary format (differentiated with *.mesh* or *.meshb* extensions).

This choice is transparent from a user's point of view and a routine reading GMF files will work on both kinds of storage. The library determining the right access method depending on the file extension.

It is advised to use ascii for debugging purpose only, when a file needs to be hand-written or checked by a human eye. Otherwise, when performance, compactness and portability are of concerns, binary is the way to go.

1.4.1 Size does matter

Binary files have a slightly smaller footprint than their ascii counterparts (typically 30% less). Not only does it save space on hard drives, but it allows for faster transfer as well.

1.4.2 About performance

Great care has been taken on performance issues when creating the *libMeshb*. When dealing with binary files, reading and writing throughputs will only be limited by the speed of the physical media where those files are located. Speed ranging from 20 MB/s to 60 MB/s can be achieved with hard drives and 10 MB/s or 100 MB/s with fast ethernet and gigabit ethernet networks, respectively.

The *libMeshb* performs very poorly in ascii mode, which is more processor bound rather than hard-drive bound. Don't expect more than 5 or 10 MB/s throughputs.

1.4.3 Compatibility issue: little vs. big endian

When it comes to binary storage, the compatibility problem posed by endianness always comes to mind.

Some processors like PowerPc or SPARC are called big endian because of the way they store bytes within a word from most significant byte to the lowest.

The other ones, like i86 (Intel core2, AMD opteron) or itanium, store bytes in the reverse order and are called little endian.

Consequently, a binary word written by a big endian processor cannot de read by a little endian one, and vice versa. This problem can be easily overcome by reversing bytes order when reading misoriented data. The *libMeshb* handles this compatibility issue via a control word that indicates which endian a mesh file was written in.

You may then use binary files as safely as ascii ones.

1.5 Mesh and Sol files

For the sake of understanding, different extensions must be given to files containing mesh related keywords, *.mesh* or *.meshb*, and files containing physical solution keywords, *.sol* or *.solb*.

1.6 File versions

Over the years, the library had to adapt to ever increasing system capabilities, henceforth, modification to the binary file format had to be done. As of today, there are three revisions of the meshb format:

Version	Size of integers	Size of reals	Maximum size of file
1	32 bits	32 bits	2 Giga Bytes
2	32 bits	64 bits	2 Giga Bytes
3	32 bits	64 bits	8 Exa Bytes
4	64 bits	64 bits	8 Exa Bytes

Although the *libMeshb* still handles versions 1 and 2 for the sake of compatibility, it is strongly advised to create version 3 files since more and more computer are 64-bit capable.

A word of caution: great care must be taken when setting the library's arguments type. Regardless of the file version, some arguments are mandatory 32 bits integer like the open mode tag, mesh dimension or the file index. Even in "full 64 bits" version 4 mesh file format, only the number of lines given or set by the GmfSetKeyword() of GmfStatKeyword() commands and vertex indices used by elements field are using 64 bits integers.

The 64 bits integer data type used by the library is the *long int*, which may be set to 32 bits by come compilers. In this case, the *libMeshb* will simply return with an error when dealing with version 4 mesh files. Most compilers, GNU and Intel for instance, set the long int size to 64 bits unless you specify the "-m32" option.

2 Quick-start

This section will guide you through three simple examples from which you may easily cut and paste to build your own code.

2.1 Reading a file

Let's start with reading an existing mesh file.

Reading a mesh is a two-step scheme:

1. opening and checking the file and allocating data structures according to its content
2. reading fields of interest (vertices, elements, etc...) and storing them in the previously allocated structures

2.1.1 Open, check and allocate a mesh

Opening a mesh file is done via the GmfOpenMesh() command. It allows to check for file existence and whether it is of the required version and dimension.

```
long long MeshIndex;
int Version, Dimension;
MeshIndex = GmfOpenMesh("testcase.meshb", GmfRead, &Version, &Dimension );
```

Then, the presence and quantity of each item can be checked and memory allocated accordingly via the GmfStatKwd() command.

```
int NumberOfTriangles, (*TableOfTriangles)[4];

NumberOfTriangles = GmfStatKwd( MeshIndex, GmfTriangles );

if(NumberOfTriangles > 0)
    TableOfTriangles = malloc(NumberOfTriangles * 4 * sizeof(int));
```

2.1.2 Example: reading vertices and triangles

Reading each keyword data is done via two commands:

- GmfGotoKwd() to set the file index to the beginning of keyword data
- GmfGetLin() to read one line of data

Let's say we would like to open a file, check if it contains vertices and quads, and read those fields into their respective tables:

```
long long idx;
int ver, dim, nbt, (*tt)[4], nbv, *rt;
float (*ct)[3];

/* Try to open the file and ensure its version is 1
   (single precision reals) and dimension is 3 */

idx = GmfOpenMesh("tri.meshb", GmfRead, &ver, &dim );

if( !idx || (ver != 1) || (dim != 3) )
    exit(1);

/* Read the number of vertices and triangles and allocate
   a triangle table (tt[nbt][4]) to store each triangle
   vertices and reference (hence the fourth integer).
   Two tables are allocated for the vertices:
   ct[nbv][3] to store the three coordinates
   rt[nbv] to store the references. */

nbv = GmfStatKwd( idx, GmfVertices );
nbt = GmfStatKwd( idx, GmfTriangles );

if( !nbv || !nbt )
    exit(1);

tt = malloc( nbt * 4 * sizeof(int) );
```

```
ct = malloc( nbv * 3 * sizeof(float) );
rt = malloc( nbv * sizeof(int) );

/* Move the file pointer to the begining of vertices data
   and start to loop over them. Then do likewise with triangles. */

GmfGotoKwd( idx, GmfVertices );

for(i=0;i<nbv;i++)
  GmfGetLin( idx, GmfVertices, &ct[i][0], &ct[i][1], &ct[i][2], &rt[i]);

GmfGotoKwd( idx, GmfTriangles );

for(i=0;i<nbt;i++)
  GmfGetLin( idx, GmfTriangles, &tt[i][0], &tt[i][1], &tt[i][2], &tt[i][3] );

GmfCloseMesh( idx );
```

2.2 Writing a file

Writing a mesh is also a two-step scheme:

1. creating a empty mesh file with the right version and dimension
2. writing every fields (vertices, elements, etc...)

2.2.1 Creating and defining a mesh

Mesh name, version and dimension must be provided at creation time. Creating a mesh following version 1 (single precision real numbers) in three dimensions named testcase.meshb runs this way:

```
Meshindex = GmfOpenMesh( "testcase.meshb", GmfWrite, 1, 3 );
```

2.2.2 Example: writing vertices and triangles

Following the reading example, we would like to write back the data to a new file:

```
long long idx;
int ver, dim, nbt, (*tt)[4], nbv, *rt;
float (*ct)[3];

/* Try to create a three-dimensional, version 1
   (single precision reals) file */
```

```
idx = GmfOpenMesh("tri.meshb", GmfWrite, 1, 3 );

if( !idx )
    exit(1);

/* Setup a vertex field with nbv lines
   and loop over vertices to write them down.
   Note that this time, direct values are passed on
   GmfSetLin() instead of pointers. */

GmfSetKwd( idx, GmfVertices, nbv );

for(i=0;i<nbv;i++)
  GmfSetLin( idx, GmfVertices, ct[i][0], ct[i][1], ct[i][2], rt[i]);

GmfSetKwd( idx, GmfTriangles, nbt );

for(i=0;i<nbt;i++)
  GmfSetLin( idx, GmfTriangles, tt[i][0], tt[i][1], tt[i][2], tt[i][3] );

GmfCloseMesh( idx );
```

2.3 Doing it all together

In this last example, the file "quad.mesh" a three-dimensional mesh made of quads will be read, transformed into a triangulated one, which will be written as "tri.mesh":

```
long long InpMsh, OutMsh;
int i, nbv, nbq, ver, dim, *rt, (*qt)[5];
float (*ct)[3];

/* Open and check the input quadrilateral mesh */

InpMsh = GmfOpenMesh("quad.mesh", GmfRead, &ver, &dim);

if( !InpMsh || (ver != 1) || (dim != 3) )
  exit(1);

/* Allocate vertices and quads tables */

nbv = GmfStatKwd(InpMsh, GmfVertices);
ct = malloc(nbv * 3 * sizeof(float));
```

```
rt = malloc(nbv * sizeof(int));

nbq = GmfStatKwd(InpMsh, GmfQuadrilaterals);
qt = malloc(nbq * 5 * sizeof(int));

/* Read vertices and quads then close the input file */

GmfGotoKwd(InpMsh, GmfVertices);

for(i=0;i<nbv;i++)
  GmfGetLin(InpMsh, GmfVertices, &ct[i][0], &ct[i][1], &ct[i][2], &rt[i]);

GmfGotoKwd(InpMsh, GmfQuadrilaterals);

for(i=0;i<nbq;i++)
  GmfGetLin(InpMsh, GmfQuadrilaterals, &qt[i][0], \
            &qt[i][1], &qt[i][2], &qt[i][3], &qt[i][4]);

GmfCloseMesh(InpMsh);

/* Now create the output file.
   Each quad being split into two triangles. */

if(!(OutMsh = GmfOpenMesh("tri.mesh", GmfWrite, ver, dim)))
  exit(1);

GmfSetKwd(OutMsh, GmfVertices, nbv);

for(i=0;i<nbv;i++)
  GmfSetLin(OutMsh, GmfVertices, ct[i][0], ct[i][1], ct[i][2], rt[i]);

GmfSetKwd(OutMsh, GmfTriangles, 2*nbq);

for(i=1;i<=nbq;i++)
{
  GmfSetLin(OutMsh, GmfTriangles, qt[i][0], qt[i][1], qt[i][2], qt[i][4]);
  GmfSetLin(OutMsh, GmfTriangles, qt[i][0], qt[i][2], qt[i][3], qt[i][4]);
}

GmfCloseMesh(OutMsh);
```

3 Commands

3.1 GmfOpenMesh

Open a mesh file for reading or writing: in reading mode, it tries to open the file and returns some information about its content, in writing mode it creates an empty mesh file.

3.1.1 Reading mode

```
long long GmfOpenMesh( char *FileName,
                       int OpenMode,
                       int *Version,
                       int *Dimension );
```

FileName: this string must contain the path and the mesh name with its extension (meshes/my_mesh.meshb).

OpenMode: must be set to GmfRead.

Version: will be set to the value read from file, which may range from 1 to 3.

1. real numbers in the whole file are written in single precision (32 bits)
2. real numbers in the whole file are written in double precision (64 bits)
3. same as 2 but file size may be greater than 2 GBytes.

Dimension: will be set to the value read from file, only dimensions 2 and 3 are supported.

Returns: Zero on failure or the opened mesh index otherwise. This index should be properly stored since it must be provided to any further *libMeshb* commands working on this file.

Example: open a mesh file and print its version and dimension.

```
long long MeshIndex;
int Version, Dimension;

Meshindex = GmfOpenMesh("testcase.meshb", GmfRead, &Version, &Dimension );

if(MeshIndex)
    printf("Version = %d, Dimension = %d\n.", Version, Dimension);
else
    puts("This file cannot be opened.");
```

3.1.2 Writing mode

```
long long GmfOpenMesh( char *FileName,
                       int OpenMode,
                       int Version,
                       int Dimension );
```

FileName: this string must contain the path and the mesh name with its extension (meshes/my_mesh.meshb).

OpenMode: must be set to GmfWrite.

Version: must be provided at file creation, see *Reading mode* for version values.

Dimension: must be provided at file creation, only dimensions 2 and 3 are supported.

Returns: zero on failure or the opened mesh index otherwise. This index should be properly stored since it must be provided to any further *libMeshb* commands working on this file.

Example: create a new three dimensional mesh file storing double precision numbers.

```
Meshindex = GmfOpenMesh("newfile.meshb", GmfWrite, 2, 3 );
```

3.2 GmfCloseMesh

A mesh file must be properly closed in order to release any allocated memory and to write tailing information.

```
GmfCloseMesh( long long MeshIndex );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided for the file to be closed.

3.3 GmfStatKwd

This command queries the mesh file for the presence of a given keyword and the number of associated lines.

3.3.1 Getting information on a mesh keyword

```
int GmfStatKwd( long long MeshIndex,
                int Keyword );
```

MeshIndex: index of referenced mesh.

Keyword: the keyword tag you are requesting information on (see section 4 for a full list of available keywords).

Example: check out and print the number of triangles in a mesh file.

```
int NumberOfTriangles;

NumberOfTriangles = GmfStatKwd( MeshIndex, GmfTriangles );

if(NumberOfTriangles)
    printf("This file contains %d triangles\n.", NumberOfTriangles);
else
    puts("This file does not contain any triangle.");
```

3.3.2 Getting information on a solution keyword

In this case, additional information will be provided: the number of fields per solution, the number of real numbers a solution line occupies and a table of solutions types.

```
long long GmfStatKwd( long long MeshIndex,
                      int Keyword,
                      int *NumberOfTypes,
                      int *SizeOfSolution,
                      int *TableOfTypes );
```

MeshIndex: index of referenced mesh.

Keyword: the keyword tag you are requesting information on (see section 4 for a full list of available keywords).

NumberOfTypes: pointer to an integer, it will be set to the number of fields in the solution.

SizeOfSolution: pointer to an integer, it will be set to the number of real numbers (float or double depending on the file version) used by a solution line for memory allocation purpose.

TableOfTypes: pointer to a previously allocated table which will be filled with the type of each solution field.

Example: check out and print the number of solutions and their kinds associated to vertices.

```
int NmbSol, NmbTypes, NmbReals, TypesTab[ GmfMaxTyp ];

NmbSol = GmfStatKwd( MeshIndex, NmbSol, &NmbTypes, &NmbReals, TypesTab );

if(NmbSol)
{
  printf("This file contains %d solutions at each vertex\n.", NmbSol);
  printf("Each solution contains %d fields:\n", NmbTypes);

  for(i=0; i<NmbTypes; i++)
  {
    switch(TypesTab[i])
    {
      case GmfSca    : printf("scalar,\n"); break;
      case GmfVec    : printf("vector of %d scalars,\n", dim); break;
      case GmfSymMat : printf("upper triangular part of a symetric \
                       %d x %d matrix,\n", dim, dim); break;
      case GmfMat    : printf("full %d x %d matrix,\n", dim, dim); break;
    }
  }
}
else
    puts("This file does not contain triangles.");
```

3.4 GmfGotoKwd

Prior to reading each line of a keyword with the GmfGetLin() command, the file position must be set to the beginning of its data with GmfGotoKwd(). Note that positioning the file mark is only needed when reading, not writing.

```
int GmfGotoKwd( long long MeshIndex, int Keyword );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose data are to be read.

Returns: zero if this keyword is not present in the pointed file, one otherwise.

3.5 GmfSetKwd

Prior to writing each line of a keyword with the GmfSetLin() command, the keyword header should be written along with the number of lines.

3.5.1 Writing a mesh keyword

```
int GmfSetKwd( long long MeshIndex,
               int Keyword,
               int NumberOfLines );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose data are to be written.

NumberOfLines: number of data lines which are to be written.

Returns: zero if the data could not be written, and the number written lines otherwise.

3.5.2 Writing a solution keyword

When it comes to solution keywords, two extra arguments must be passed on: a table of solution types and its size.

```
int GmfSetKwd( long long MeshIndex,
               int Keyword,
               int NumberOfLines,
               int NumberOfTypes,
               int *TableOfTypes );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose data are to be written.

NumberOfLines: number of data lines which are to be written.

NumberOfTypes: the number of fields stored for each line of this solution. It sets the size of the following TableOfTypes containing each field type.

TableOfTypes: pointer to a table of integers, each entry setting the type of each solution field: 1 for a scalar, 2 for a vector, 3 for symmetric matrix and 4 for a full matrix.

Returns: zero if the data could not be written, and the number written lines otherwise.

3.6 GmfGetLin

GmfGetLin() is a variable arguments command, it reads one line of data from the file and stores each items in the provided pointers to user's data structures.

```
int GmfGetLin( long long MeshIndex,
               int Keyword,
               arguments );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose line data is to be read.

arguments: as many pointers to the required type of data as stated by the keyword definition (see section 4) should be provided.

Example: reading a vertex in three-dimensional case. Caution: the right size of real numbers, float or double, should be provided according to the mesh file version.

```
int ref;
float xf, yf, zf;
double xd, yd, zd;

if(Version == 1)
    GmfGetLine(MeshIndex, GmfVertices, &xf, &yf, &zf, &ref);
else
    GmfGetLine(MeshIndex, GmfVertices, &xd, &yd, &zd, &ref);
```

3.7 GmfSetLin

This commands works pretty much like GmfGetLin(), but arguments are given directly instead of pointers.

```
int GmfSetLin( long long MeshIndex,
               int Keyword,
               arguments );
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose line of data is to be written.

arguments: as many values of the required type of data as stated by the keyword definition (see section 4) should be provided.

3.8 GmfGetBlock

GmfGetBlock() is a variable arguments command, it reads all the lines of data from the file and stores each items in the provided pointers to user's data structures. The user's data structure has to be fully described in order for the library to fill all the lines automatically.

```
int GmfGetBlock(long long MeshIndex, int Keyword, long long BeginLine,
                long long EndLine, Procedure, arguments...);
```

MeshIdx: the index returned by GmfOpenMesh() must be provided.

KeyWord: code of the keyword whose lines of data are to be read.

BeginLine: starting line in the mesh file, it enables partial reading for parallelism.

EndLine: ending line in the mesh file.

Procedure: pointer to an optinal user's procedure that will be called in parallel after each block has been read. If a procedure is given, a second pointer on user's data must be provided right after.

arguments: for each type of data as stated by the keyword definition (see section 4), three arguments must be provided. First, the user's type of data in which the file's data will be stored (four kinds are available: GmfFloat, GmfDouble, GmfInt and GmfLong). Second, a pointer to the first line of this data type in the user's structure. Third, the same pointer but on the last line. The example below is more telling.

Example: reading all vertices in three-dimensional case.

```
int ref[nv];
double x[nv], y[nv], z[nv];

GmfGetBlock(MeshIndex, GmfVertices, 1, nv, NULL, \
            GmfDouble, &x[1],   &x[nv], \
            GmfDouble, &y[1],   &y[nv], \
            GmfDouble, &z[1],   &z[nv], \
            GmfInt,    &ref[1], &ref[nv]);
```

3.9 GmfSetBlock

Works exactly as GmfGetBlock exept that all lines are written, you cannot specify starting and ending lines in the file since concurrent writing is not supported for now. Note that you still need to set the keyword first with the help of GmfSetKwd() prior to writing the whole data lines with GmfSetBlock().

Example: applying a preprocesing function on vertices before writing them on disk.

```
int ref[nv];
double x[nv], y[nv], z[nv];

GmfSetBlock(MeshIndex, GmfVertices, FlipRefs, ref, \
            GmfDouble, &x[1],   &x[nv], \
            GmfDouble, &y[1],   &y[nv], \
            GmfDouble, &z[1],   &z[nv], \
            GmfInt,    &ref[1], &ref[nv]);

FlipRefs(long long begin, long long end, void *data)
{
    int *ref = (int *)data;
    long long i;

    for(i=begin;i<=end;i++)
        if(ref[i] == 1)
            ref[i] = 2;
        else
            ref[i] = 1;
}
```

4 Keywords

4.1 List of basic keywords

Those are topologic and geometric data types, commonly used in meshes such as vertices, triangles or normal vectors. Consequently they can only be used in *.mesh* or *.meshb* files.

They are made of a header, indicating the keyword code and the number of data lines stored in the file, followed by as many lines as stated.

Each data line format is described in the following table:

keyword	
data	description
Comments	
1 string	each strings cannot exceed 256 characters including the trailing 0
Corners	
1 integer	vertex index: this vertex is a geometric corner
Edges	
3 integers	vertex indices and a reference
Hexahedra	
9 integers	vertex indices and a reference
HexahedraP2	
28 integers	vertex indices and a reference
Normals	
2 or 3 reals	normal vector: 2 or 3 components depending on the mesh dimension
NormalAtQuadrilateralVertices	
4 integers	there must be as many NormalAtQuadrilateralVertices as there are Quadrilaterals in a mesh, each NormalAtQuadrilateralVertices line pointing implicitly to the respective quad. The four integers are associated with the quad vertices, they are indices pointing to a normal in the Normals table.
NormalAtTriangleVertices	
3 integers	there must be as many NormalAtTriangleVertices as there are Triangles in a mesh, each NormalAtTriangleVertices line pointing implicitly to the respective triangle. The three integers are associated with the triangle vertices, they are indices pointing to a normal in the Normals table.
NormalAtVertices	
2 integers	first integer points to a vertex and the second one points to the associated normal vector index

Pentahedra	
7 integers	vertex indices and a reference
Quadrilaterals	
5 integers	vertex indices and a reference
QuadrilateralsP2	
10 integers	vertex indices and a reference
RequiredEdges	
1 integer	edge index: this edge is required can not be modified
RequiredQuadrilaterals	
1 integer	quad index: this quad is required can not be modified
RequiredTriangles	
1 integer	triangle index: this triangle is required can not be modified
RequiredVertices	
1 integer	vertex index: this vertex is required can not be modified
Ridges	
1 integer	edge index: this edge is a ridge (geometric sharp angle)
Tangents	
2 or 3 reals	tangent vector: 2 or 3 components depending on the mesh dimension
TangentAtEdgeVertices	
3 integers	first integer points to an edge and the last two one points to the associated tangent vector indices
TangentAtVertices	
2 integers	first integer points to a vertex and the second one points to the associated tangent vector index
Tetrahedra	
5 integers	vertex indices and a reference
TetrahedraP2	
11 integers	vertex indices and a reference
Triangles	
4 integers	vertex indices and a reference
TrianglesP2	
7 integers	vertex indices and a reference
Vertices	
2 or 3 reals + 1 integer	vertex coordinates followed by a reference

4.2 List of solution keywords

Those keywords are computation related and are to be used in *.sol* or *.solb* files.

They are made of an extended solution header and multiple data lines.

The header is similar to its mesh counterpart, but adds a solution format table to describe the number of fields and their types (scalar, vector or matrix) associated with each mesh entity.

There are basically two ways to store solutions associated with a mesh:

- Direct way. SolAtElement like keywords store data fields directly associated with each element.
- Indirect way. At first, data are directly stored for each vertex via the DSolAtVertices keyword. Then, ISolAtelements like keywords will have each element vertices pointing indirectly to a DSolAtVertices solution.

keyword	
data	description
DSolAtVertices	
SolSize * reals	as many reals as stated in the DSolAtVertices keyword header
ISolAtEdges	
2 integers	there must be as many ISolAtEdges as there are Edges in a mesh, each ISolAtEdges line pointing implicitly to the respective edge. The two integers are associated with the edge vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtHexahedra	
8 integers	there must be as many ISolAtHexahedra as there are Hexahedra in a mesh, each ISolAtHexahedra line pointing implicitly to the respective hex. The eight integers are associated with the hex vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtPentahedra	
6 integers	there must be as many ISolAtPentahedra as there are Pentahedra in a mesh, each ISolAtPentahedra line pointing implicitly to the respective penta. The six integers are associated with the penta vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtQuadrilaterals	

4 integers	there must be as many ISolAtQuadrilaterals as there are Quadrilaterals in a mesh, each ISolAtQuadrilaterals line pointing implicitly to the respective quad. The four integers are associated with the quad vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtTetrahedra	
4 integers	there must be as many ISolAtTetrahedra as there are Tetrahedra in a mesh, each ISolAtTetrahedra line pointing implicitly to the respective tet. The four integers are associated with the tet vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtTriangles	
3 integers	there must be as many ISolAtTriangles as there are Triangles in a mesh, each ISolAtTriangles line pointing implicitly to the respective triangles. The three integers are associated with the triangle vertices, they are indices pointing to solutions fields in the DSolAtVertices table.
ISolAtVertices	
1 integer	there must be as many ISolAtVertices as there are Vertices in a mesh, each ISolAtVertices line pointing implicitly to the respective vertex. The integer is an index pointing to a solutions field in the DSolAtVertices table.
SolAtEdges	
SolSize * reals	as many reals as stated in the DSolAtVertices keyword header
SolAtHexahedra	
SolSize * reals	as many reals as stated in the SolAtHexahedra keyword header
SolAtPentahedra	
SolSize * reals	as many reals as stated in the SolAtPentahedra keyword header
SolAtQuadrilaterals	
SolSize * reals	as many reals as stated in the SolAtQuadrilaterals keyword header
SolAtTetrahedra	
SolSize * reals	as many reals as stated in the SolAtTetrahedra keyword header
SolAtTriangles	
SolSize * reals	as many reals as stated in the SolAtTriangles keyword header
SolAtVertices	

SolSize * reals	as many reals as stated in the SolAtVertices keyword header

4.3 Miscellaneous keywords

Finally, those basic keywords have no header and contain only one line of data, most often giving global information on the mesh or the solution file.

keyword	
data	description

AngleOfCornerBound	
1 real	threshold angle for automatic sharp features detection, in degrees

BoundingBox	
4 or 6 reals	the box coordinates bounding the whole mesh: x_{min}, x_{max}, y_{min}, y_{max} and z_{min} and z_{max} (in three dimensional case only)

Iterations	
1 integer	discretionary iteration counter

Time	
1 real	discretionary time counter