LifeV Developer Manual

Revision: 1.10, October 9, 2009, 13:29:51 UTC Printed: April 13, 2017

G. Fourestey
S. Deparis
L. Formaggia
J.-F. Gerbeau
C. Prud'homme

April 13, 2017

This manual is for LifeV (version 0.1, 8 March 2004), a library for scientific computing specially aimed at fluid-structure interaction and blood flow simulation.

Contents

List of Figures

List of Tables

Chapter 1

Program Development Conventions

1.1 Documentation Rules

All classes and methods should have documentations lines which explain in a concise but thorough way it usage. Documentation should be provided with methods *declarations*

The amount of information contained in the documentation of functions and methods should augment in the following order (unless there are special needs)

- Inlined private methods, constructors and destructor;
- Inlined methods;
- Public methods;
- User interface methods.

By *user interface methods* we intend those methods that are meant to be directly called by the user of the library. Those methods should be the best documented.

1.1.1 Methods and function documentation

I present a template for documentation of a method, which is meant to serve as example.

```
void Euler::Gudonov(Real nflux[4], Real const lsol[4],
                    Real const rsol[4], Real const & nx,
                    Real const & ny)
{
/*
  #Version  1.0. Released 1/1/99. Marco Manzini 1999

  #Purposes: Computed Gudonov numerical fluxes
   (This part describes what the routine does)

  #Input: lsol  -> left state   (density, ux, uy, pressure)
         rsol  -> right state  (,,)
         nx,ny -> references to the
                  outward oriented components of normal of control volume
   (This part describes the input
```

```
  #Output nflux -> numerical fluxes

  #Preconditions: lsol and rsol, should be valid states.
                  nx*nx + ny*ny =1

  #Preconditions Tests
                  Euler::ok.state should be used for the validity check of
                  lstate and rstate.


  #Postconditions: nflux returns the numerical flux according to Gudunov
                   method.

  #Postcondition Tests Not provided


*/
```

In the documentation we provide a few keywords, identified by `#`.

The *Version* keyword introduce a comment line which contains a reference number and *DATE* which could be used to identify the revision of the routine we are using. This is particularly important for functions whose implementation may change. In particular, The keywords *experimental*, and *validated* should be respectively used to indicate software still on the experimental stage (no extensive validations made), and software already validated by tests. The name of the principal author should appear as well.

The *Purposes* keyword introduces a brief description of the function scopes. Without establishing hard rules, the details contained in this part should increase the more the function usage is complex, the more it is near to the "final user" (in particular a detailed description should be done of user interface functions). In addition, it is expected that this part should be more carefully written in case of *validated* software than *experimental* one.

The *Input* keyword introduces the description of the input arguments. A constant input argument is an argument which is *never* modified by the function. Whenever a constant input is passed by reference it *should be indicated by the C++ keyword* `const`*, i.e. it should be made a constant reference*. This help both the user and the compiler. Some routines may take additional input data from a file (or in general from an input stream). In that case, details should be indicated in this section.

The *Output* keyword introduces a brief description of the function outputs, which may be manifold:

1. A non-constant reference argument (note that a non-constant reference may be also an input, in that case its description will be repeated in the input section);
2. A return value for the function;
3. An output stream.

The *Preconditions* keyword contains the conditions the input should satisfy in order to be proper. In particular the *Preconditions Tests* section contains the lists of methods which should be used to verify if the preconditions are satisfied. The preconditions test *should be automatically activated if the compiler switch* `-DTEST_PRE` *has been used* (see section on debugging).

The *Postconditions* keyword contains the conditions the output should satisfy in order to be proper.In particular the *Postconditions Tests* section contains the lists of methods which

should be used to verify if the postconditions are satisfied. The postconditions tests *should be automatically activated if the compiler switch* `-DTEST_POS` *has been used* (see section on debugging).

Array bounds checking is switched on by the `TEST_BOUNDS` switch.

1.1.2 Class documentation

```
class FiniteElement : public GenericFiniteElement
{
/*
  #Version  1.0. Released. Giulio Cesare 89 A.C.

  #Purposes: This class contains an implementation of the SPQR finite
             elements.

  #Public data:

  #Private data:

  #Invariants

  #Invariant Tests


*/
```

The keyword *version* has the same meaning as for the case of a function.

Purposes is used to introduce a brief but exhaustive description of tha class scopes.

Public Data introduces a description of any public data which the class contains, in particular static data.

The keyword *Private data* should contain a description of the private data. It is used to give information to a potential programmer of the class. Thus it is required only for classes which are likely to be modified and for public data whose description if felt necessary for a clear understanding of the algorithms implemented in the classes method.

The keyword *Invariants* tags the description of the class invariant quantities. These are properties that the class public and private data should satisfy if the class is in a correct state. For instance, one may impose that each instance of the class *Positive_Definite_Matrix*, used for positive definite matrices, should satisfy the condition that the minimum eigenvalue of the stored matrix is positive.

One may than devise a test, which will be a class method and will be described under the sub-item *Invariants tests*, which verify that condition. The test should be used for debugging purposes and activated when the compiler flag `-DTEST_INV` is switched on (see section on debugging).

1.1.3 File documentation

Since class and methods are declared in a `.h` file, the first line of the file will contain as well some of the documentation. In particular the *Version* and the *Purpose* field. For the *Version* field, we will follow the convention that

- If a class does not contain it, the value of the file containing the class is used;

- If a function description does not contain it and the function is a method, the one of the class applies, otherwise that of the file containing the function declaration.

1.2 General Nomenclature and Programming Convention

1.2.1 Coding Standards

We will follow the following simple rules:

- All declaration in `*.h` files. Method definitions (a part inlined methods or other possible special cases) are in a `cc` file with the same name;
- No inlined methods definition within classes declarations, unless they are **VERY** short. Inlined methods will be defined immediately after class declaration.
- Use `INLINE` macro, instead of `inline`. The inline macro is switched off if `-DNOINLINE` is used.
- Variables and classes must have names which recall their usage. Public variables named `a` or `pippo` are forbidden.
- Class name have their first letter UPPER case, if the name is formed by many "words" the word initial letter is also upper case. Ex.: `class MySimpleArray`.
- *Public* variables and functions names follow the same rule as class names, a part from teh fact that the first letter is *always* lower case. Example: *Real pressure*, `Matrix & computeMassMatrix(FiniteElement const & localElement)`
- Avoid function declarations without argument name. That is

  ```
  Matrix & computeMassMatrix(FiniteElement const & localElement);
  ```

  should be used instead of

  ```
  Matrix & computeMassMatrix(FiniteElement const & );
  ```

  This greatly helps understanding how the function works.
- Use `const` keyword when possible. It helps the compiler and the human being reading the code. Moreover, it enhance debugging.
- Typedefs aliases name follow the same rule as classes names.
- Use the typedefs aliases `Real`, `Int` and `Uint`, instead of the in-built types `float`, `Int` and `unsigned int`. It helps making code changes afterwards.
- C preprocessor macros name are ALL UPPERCASE.
- All principal classes will have a method

  ```
  void  showMe(ostream & logStream);
  ```

  which output on `logStream` (which may be the standard output or a file) the state of the class (i.e. the content of the global and local variables. Example (to be verified)

```
class AStupidArray {
public:
void addToArray(Real const & value);
...
void  showMe(ostream & logStream);
private
_myArray[4][4];
}
void AStupidArray::showMe(ostream & logStream)
{
for(int i =0; i<4; ++i)
 {
   for(int j =0; j<4; ++i)
    logSteam << ''_myArray['' << i << '','' << j << ''] ''<<_myArray[i][j]<< endl;
 }
}
```

We will also folly the coding practices outlined in the document **Coding Standards** elaborated by *The Laboratory for Scientific Computing, University of Notre Dame*, available searching the Lab home page at `http:///www.lsc.nd.edu`.

1.3 Static and dynamic polymorphism

1.4 Standard Template Library

1.4.1 Containers

We will make extensive use of STL containers (in particular vectors and maps). This especially in the mesh handler.

1.4.2 Strings

We will use STL strings classes, much more flexible than C-style strings.

1.4.3 valarray

`valarray` class is still not available in gnu c++. Then we will avoid it for the moment. Yet, we will consider its adoption when correctly implemented in g++.

1.5 Namespaces

Still too early, but we suggest to read the C++ reference manual. They may be of help to avoid name clashing in the future.

1.6 Debugging and assertions

1.7 Debugging

Debugging is essential! In the following we address the following issues

- Usage of `asserts`;
- Usage of `cpp -D` compiler switch for switching on/off debugging at different level;

- Usage of `cpp defines` to implement useful macros and reduce typing (and typing errors!);
- Debugging inlined functions.

1.7.1 Assertions

Assertion is a way of control the satisfaction of certain conditions. If the condition is not satisfied an error message is printed and the program is aborted.

The C header file `assert.h` provide some basic utilities for assertion, in particular the command `assert(b)`, will abort the program if $b = 0$ (i.e. false) giving an indication of the file and the line where the assertion has failed. The `assert` is in fact a `cpp` macro which can be turned off by setting the compiler switch `-DNDEBUG` (NOT debug). It is rather crude, thus we propose something similar but more sophisticated, following what has been done by E.Berolazzi and M.Manzini in the `P2MESH` Library. We define the following set of macros:

`ERROR_MSG(A)`	Basic error message handler, used by all other macros. It is NEVER switched off. It prints the stream `A` (which may be a string of something more complex) on the standard error `cerr` stream, lus an indication of the line and file where the condition occurred, and aborts the program.
`ASSERT(X,A)`	Verifies condition `A` (which will be a logical expression), if not satisfied it will abort the program, writing the stream `A` plus an indication of the line and file where the condition occurred. It is switched **off** ONLY if `-DNODEBUG` is used at compile level. This should be used for the basic error checks that we would normally let active.
`ASSERT_PRE(X,A)`	Specialised `ASSERT` for *preconditions* (see documentation section). It is switched **on** ONLY if `-DTEST_PRE` of `-DTEST_ALL` is used at compilation time.
`ASSERT_POS(X,A)`	Specialized `ASSERT` for *postconditions* (see documentation section). It is switched **on** ONLY if `-DTEST_POS` of `-DTEST_ALL` is used at compilation time.
`ASSERT_INV(X,A)`	Specialized `ASSERT` for *invariants* (see documentation section). It is switched **on** ONLY if `-DTEST_INV` of `-DTEST_ALL` is used at compilation time.

The last three asserts allow to modulate testing of pre, post conditions and invariants. `ASSERT` macro have a `0`ed version (e.g. `ASSERT0(X,A)`) which is never turned off. An example

```
bool complicatedCondition(const Matrix & A);
.....
.....
ASSERT(a>0,''The quantity a should be positive. Instead it is equal to ''<< a) ;
....
....
ASSERT_PRE(complicatedCondition(B),''Precondition test on matrix B not satisfied'') ;
...
...
```

The `ASSERT`'s macros (c preprocessor macros) are defined in the `lifeV.h` file. (YES, until somebody else give me another name, I will call the software *lifeV*, in memory of the all *life*s experienced in the past (I skipped III and IV, just for fun).

1.7.2 Compiler Switches and define's

- `-DTEST_BOUND` switches on bound checking in all arrays-like structure used in the library.
- `-DTEST_PRE` Preconditions are checked.
- `-DTEST_INV` Invariants are checked.

- `-DTEST_POS` Postconditions are checked.
- `-DNOINLINE` switches of all `INLINE` macros. `INLINE` macros should be used in place of `inline`, in order to be able to switch off in-lining during debugging (in-lining makes debugging more complicated).
- `-DTEST_ALL` Equivalent to `-DTEST_BOUND -DTEST_PRE -DTEST_INV -DTEST_POS -DNOINLINE`
- `-DSAVEMEMORY` Does not explicitely store some arrays, but rebuild the information each time needed. It saves some memory at the expense of computional time.
- `INT_BCNAME` Uses as name for BConditions just an integer and not a string. It saves memory at the expense of a more cryptic program.

1.8 Type nemas, typedefs and template arguments

It is very important to follows the convention for Type and Template Arguments names. C++ allows to give alias to typenames with the instruction `typedef` and this is a very useful practice, in particular if **a common set of names is constantly used** which correspond to a clearly identified set of classes. Before recalling the main Types and setting up their general use we wish to recall some basic facts.

• Public typedefs are inherited.
Example

```
 class a {
public:
        typedef float Real;
 ....
};

class b: public a
{
....
};
```

Now, `b::Real` is in fact a `float`!

• It is a good practice to expose with a typedef the main template arguments of a class template.
Example:

```
template<typename GEOSHAPE>
class AFiniteElement{

public:
        typedef  GEOSHAPE GeoShape;
........
};
```

So that it os possible to do

```
typedef AFiniteElement<Tetra> ATetraElement;


int main()
{
        ATetraElement pippo;
```

```
        ATetraElement::GeoShape theShape;
}
```

In this way the user does not have to remember the actual definition of `ATetraElement` to access the typename of the Basic Geometric Shape.

Remeber that a special care should be used when a templete class is derived by a template argument:

```
class ABaseClass
public:
typedef Triangle GeoShape;
typedef std:complex<float> Complex;
...
};

template<typename BASE>
class Derived: public BASE
{
public:
  typedef BASE Base; // I expose the templete argument

// I need this since GeoShape is USED in the definition of this class

  typedef typename BASE::GeoShape  GeoShape;
  Geoshape a; // Here I use GeoShape
};

int main()
{
 Derived<ABaseClass> pippo;
 Derived<ABaseClass>::Complex a; //OK
 Derived<ABaseClass>::GeoShape b; //OK
 .....
}
```

I do not need to make visible the type `Complex` inside the `Derived<BASE>` class templete since it is not used in the definition (and I can rely to have it available by inheritance when I instantiate the template class), I need instead to use the `typedef typename` declaration for `GeoShape` since it is used inside the definition of `Derived<BASE>`.

SONO QUI

complicated by the fact that often the template argument name is the same of the tepedef I would like to expose. For instance (an example ...) this WILL NOT WORK

template ¡typename GeoShape¿ class FE typedef GeoShape Geoshape; ;

since the typedef will be ambiguous. One will need then change it into something like

template ¡typename GEOSHAPE¿ class FE typedef GEOSHAPE Geoshape; ;

now it will work and inside the class template definition I can use GEOSHAPE or GeoShape, more or less equivalently (even if, from a logical point of view, they are different beasts).

We may decide to indicate all template arguments all upper case, yet this will require too may changes... so I will just change the ones which are needed to be changed. I prefer to change the name of the template argument while having the name of the typedef conforming to the general rules. After all the template argument name is a dummy, while it is important to be consistent with the type names!.

To make myself clearer (I hope!) a GeoShape (which is the typename for a generic Basis Geometric Shape: LinearTriangle for example) will alway typedef eather as GeoShape (generic

name) or FaceShape, EdgeShape etc (specific names) and NOT as My_Geoshape or GeoShapeType or something like that. However, I will momentarily keep (for consistency reason) also these 'old' typedefs, so to avoid too many incompatibilities. Yet they should gradually go away.

d) There is a clear difference of style between my programming and that of Jean Fred (who is more experience in C++ than I, after all). I use a lot inheritance as a mechanism of building more complex classes from simpler ones. He prefers typedef containement.

I.E.

I would have written (is just an example, it does not correspond to an actual class of the library!!!)

template¡BASISFCT,GEOMAP,Quadrule¿ class Fe: public BASISFCT, public GEOMAP public: typedef BASISFCF BasisFct; typedef GEOMAP GeoMap;

... etc etc .... ;

while jean Fred prefers

template¡BASISFCT,GEOMAP,Quadrule¿ class Fe public: typedef BASISFCF BasisFct; typedef GEOMAP GeoMap;

... etc etc .... ;

and then access BasisFct and GeoMap interfaces through the typedef. In the first case instead, the typedef statement is used only to expose the template arguments, while the interfaces of BasisFct and GeoMap are directly available through inheritance.

We may argue for months to what is best and when it is best to use one strategy or the other... In any case, the existance of both styles in the library MAKES EVEN MORE IMPORTANT to have a coherent strategy for type names and typedefs!!!!!

Chapter 2

General Program Overview

2.1 General Typedefs

Some general type definitions for the library are defined in the file `lifeV.h` and given a global scope. They are

- `Real`. Float type used for all real variables in the code. Normally aliased to `double`.
- `UInt`. Integral type that holds non negative values and in particular the size of containers. Aliased to `size_t` from the Standard Template Library.
- `ID` Integral type to use for holding the *identifier* of a geometric entity (for instance, a Point, a Face etc.). By convention, an identifier of an active entity is always supposed to *start from 1*. An active geometric entity is a entity which belongs to a *mesh*. The value of the identifier of an active geometrical entity **does** correspond the one plus the location of the entity in the *entity container*.

2.2 Nomenclature

A finite element library is a complex piece of software. It is therefore imperative to set a clear and possibly unambiguous nomenclature.

Nomenclature List

To start with, we will make a clear distinction between *Geometric Entities*, that is an entity which carry data concerning the geometry, a *Finite Element Entity*, which is an entity which carry the data more properly related to a finite element definition and *Linear Algebra Entities*, which are linked to the build up, storage and manipulation of the final linear system.

2.2.1 Nomenclature List

Geometric Entity is an object that carries data concerning the geometry. Geometric entities are `Point`, `VertexVertex`, a `Edge`, `Face` and `Volume`.

Geometry Element. The geometry item of higher phisical dimension: a `Volume` in 3D and a `Face` in 2D.

Reference Finite Element, is an object carry the data more properly related to a finite element definition, in the reference space.

Mapping, an object that carries data concerning the mapping from a `Reference Finite Element` and the `Current Finite Element.`

Quadrature An object that carries all knots and weights for quadrature rules on different `Reference Finite Element`.

Basis Finite Element. The union of a `Reference Finite Element`, a `Mapping` and a `Quadrature` form a `Basis Finite Element`. This is the basis for the integration of the various integrals appearing in the weak formulation. It has the means to provide the value of the shape functions and derivatives at integration points, the Jacobian of the mapping, and other finite element related quantities once it is associated to a specific `Geometric Element`.

Boundary Element. The element (geometry of finite element) on the boundary of the computational domain.

Current finite element. Is is the result of the association of a `Geometry Element` to a `Basis Finite Element`. It makes available all the quantities needed to compute the various integrals on that element. In other words, a `Basis finite element` as such is pretty useless since it does not know the actual geometry, yet it provides the methods to compute quantities on the current element once it has been provided of the geometrical information.

Degree of Freedom (*Dof!Degree of Freedom*). It refer (in a general sense) to the coefficient of the finite element expansion for a problem variable. It may be either *scalar* or *vector*. A `Dof` may be local to the The Class `Dof` contains also the `Local-to-global` table.

Local-to-global the `local-to-global` table provides the global numbering of a `Dof`

Node. The "location" where a Dof is held. It is normally associated with a geometrical entity (but it must not be confused with it, it is indeed a Finite Element entity).

Vertex. The geometrical entity which corresponds to a vertex of a Basis Reference Shape. The number of Vertices of a Geometrical Shape is invariant from the mapping. For instance, a *Linear!Triangle* and a *Quadratic!Triangle* Triangle have both three *Vertices*.

Basis Reference Shape (*BasRefSha*). The geometrical shape of the reference element. It may be one of the following: `Point`, `Line`, `Triangle`, `Quad`, `Tetra`, `Hexa`, `Prism`.

Basis Geometric Shape (*GeoShape*). The geometrical shape of the actual finite element (for instance `QuadraticTetra`).

Basis Geometric Element (*MeshElementMarked*) . The geometric entity associated to a finite element.

Element. An ambiguous term which may refer either to a geometric entity of to the finite element proper. Therefore it should be avoided unless its significance its clear from the context.

Edge. The 1D geometric entity which corresponds to a side of a Basis Reference Shape.

Face. The 2D geometric entity which corresponds to a face of a Basis Reference Shape. In a 2D problem it represents the *Basic Geometric Element*.

Marker. An user defined attribute which may be associated to mesh geometrical entities and which is used to store and access problem related data and function, such as a boundary condition marker. By default it is just an integer. The marker handling is, at the time being, still a rapidly evolving feature.

Volume. The 3D geometric entity which corresponds to the *Basic Geometric Element* for 3D problems.

Point. The 0D geometrical entity which fully defines the actual shape of a GeoShape. The Vertices are a subset of Points. For instance, a quadratic triangle has 3 vertices and 6 Points. In the *Local Numbering* of a MeshElementMarked, the Points which are also Vertices are numbered first.

Id. Unique identifier (type `UInt`) for a MeshElementMarked and Region mesh.

Region Mesh!Mesh. The mesh on a region of the computational domain formed by basic geometric elements of the same type.

Field. The array holding the Dof values. It may be scalar or vector.

Bare Geometric Entities. Geometric entities (`BareEdge` and `BareFace`) (see section sec:bareentity) which contain only a minimal set of data and are used internally to build the list of edges and faces per Basic Geometric Element.

2.3 Paradigms

Here the main **paradigms** that we will follow

1. The `MeshElementMarked` is the key entity. All major loops will be done on the MeshElementMarked. This is a Finite Element (an possible Spectral Element) oriented work;
2. The `Dof` (degrees of freedoms) are associated to geometrical entities;
3. On each element the `Nodes` are numbered (local numbering) following the hierarchy:
   - Nodes on Vertices
   - Nodes on Edge
   - Nodes on Faces
   - Nodes on Volumes.
4. Global `Point`s numbering follow the convention:
   - Vertices first, while Points which are not not vertices have higher numbering.
5. Global numbering of the other "lower dimensional" items (like edges..) follows instead the rule
   - Boundary items first, while internal item have an higher numbering.
6. The Region Mesh always stores the Basic Geometric Elements and the Boundary Geometric Elements. For the latter, the local Vertices numbering allow to identify the direction of the outward normal. For 3D problems we follow the right-hand rule, for 2D problems the boundary is traversed leaving the domain on the lest (anti-clockwise orientation).
7. The global numbering of the degrees of freedom (Dof) follow the rule
   - the first Dof's are those associated to the Vertices (if any);
   - then those associated to the edges (if any);
   - then those associated to the Faces (if any);
   - then those associated to the Volumes (if any);

2.4 Scope of the software

The scope of the software is o provide efficient, flexible object-oriented tools for the development of finite element (and potentially spectral element) algorithms for fluid and fluid-structure interaction problems. In the long term the library should account for

- 2D and 3D domains;
- Lagrangian and mixed finite elements;
- Parametric mapping;
- Support for spectral elements;
- Support for domain decomposition approaches;
- ALE algorithms;
- Support for parallelisation;
- Simple structural models;
- Fluid Structure coupling;
- Mesh adaption.

Since the long term goal would take at least 3 man-years of work, some short term goals have been set:

- Porting to object oriented paradigm and reprogramming in C++ the existing 2D LIFE library. This task would provide useful suggestions for the 3D work and would serve as self-teaching purpose for the part of the authors without any experience in C++ coding.
- Development of the core 3D library with support for P1, P2 and iso-P2 elements and coupling with simple structural models.
- A prototype for parallelisation using existing libraries (Aztec).

This short term goals should be accomplished in the time frame of 8 months.

Details in the section **??**.

2.5 A top-bottom overview

Here we should put an overview of the library, starting from the definition of meshes and domains, down to the FEM classes

2.5.1 Domain, Regions and Meshes

Generalities

The *Computations Domain!Domain* will generally be subdivided into *sub-domains!Domain* which are associated either to a different set of equations to be solved or to different portions which will be treated independently, for instance for parallelisation purposes.

In order to avoid ambiguities we refer to *Region* to indicate a portion of the domain "homogeneous" both with respect to the set of equations to be solved and to the type of discretisation employed.

For instance, we may decide to use Hexa elements in one region and tetrahedra in another region.

To each region is then associated a *RegionMesh*, which will then be formed by a *homogeneous* set of "elements". The union of the *RegionMesh*es forms the *Mesh*

Mesh

The mesh is a container of RegionMeshes. It will also contain all the necessary tools to exchange information between the Regions. The details have still to be sorted out.

RegionMesh

A RegionMesh is a container of *geometrical* and *topological* entities, that is entities which describe the mesh geometry and its connectivity. Every RegionMesh contains a *DomainMesh* and a *BoundaryMesh*. That is, all information on the boundary geometry is accessible separately. Some geometrical entity will have a *marker*, which in the simplest case will just be an integer, which will allow to identify boundary conditions and/or different material properties. Some methods will be available to create containers (STL maps) of pointers to geometrical entities indexed by the marker value. This would allow efficient implementation of the boundary conditions.

2.5.2 Finite Elements

2.5.3 Field

A *Field* is defined on a mesh and on a BasisFinite Element. The number of degrees of freedom are associated to the various geometrical entities and the numbering convention we adopt allows for a easy dimensioning of the field vectors and to local-to-global numbering association (see section 2.3).

2.6 Main libraries

The code will consists of 3 main libraries:

1. *Mesh handler*. The scope of the library is to read a mesh description (in a format still to be discussed) and to build the data structures which handle geometry and mesh topology informations.

2. *Algebra package* This package will contain all the classes for storing matrices and vectors and the solvers. We wish to use external produced libraries, thus the following rule should be followed:

2.7 Library header files

2.7.1 The main header file

The file `lifeV.h` contains all the definitions and macros shared by all the library. In particular we report here:

```
#define ABORT() abort() // Macro for abort

// debugging macros:
# define ERROR_MSG(A)  \
   do { cerr << endl << endl << A << endl << endl ; ABORT() ; } while (0)

// assert macros

# define ASSERT0(X,A) if ( !(X) ) \
ERROR_MSG(A << endl << "Error in file" << __FILE__ << " line " << __LINE__) ;

# define ASSERT_PRE0(X,A) if ( !(X) ) \
```

```
ERROR_MSG(A << endl << "Precondition Error " << "in file " << __FILE__ \
            << " line " << __LINE__) ;


# define ASSERT_POS0(X,A) if ( !(X) ) \
ERROR_MSG(A << endl <<"Postcondition Error " << "in file " << __FILE__ \
            << " line " << __LINE__) ;


# define ASSERT_INV0(X,A)  if ( !(X) ) \
ERROR_MSG(A <<endl <<  "Invariant Error " << "in file " << __FILE__  \
          << " line " << __LINE__) ;

# define ASSERT_BD0(X)  if ( !(X) ) \
ERROR_MSG("Array bound error " << "in file " << __FILE__  \
          << " line " << __LINE__) ;
```

The compiler switches may be used to mask some of the asserts, they are `TEST_PRE, TEST_POS, TEST_INV, TEST_BOUNDS, NOINLINE` . Their meaning is evident from their name and their relation with *assertions* is detailed in section (1.1.1). We have defined also a `TEST_ALL` compiler switch which turns on all testings. ‘ The `restrict` C++ keyword is not yet supported by many compilers, therefore it it masked by default. If a compiler has it, you should indicate `-DHAS_RESTRICT`.

Some typedefs:

```
typedef double Real;
typedef int Int;
//typedef unsigned int UInt;
typedef size_t  UInt;
typedef unsigned short int USInt;
```

Since the library support 2D and 3D problems (and in principle 1D problems) we have set a switch.

```
#if defined(TWODIM) && defined(THREEDIM) && defined(ONEDIM)
#error Dont be silly. Either One, Two or Three dimensions.
#endif
#if defined(ONEDIM)
#define NDIM 1
extern const UInt nDimensions=1;
#elif defined(TWODIM)
#define NDIM 2
extern const UInt nDimensions=2;
#elif defined(THREEDIM)
#define NDIM 3
extern const UInt nDimensions=3;
#else error You MUST compile with either -DONEDIM, -DTWODIM of -DTHREEDIM set, sorry.
#endif
```

The 1D version is, at the moment, there only “virtually”, since no 1D finite elements have been programmed yet.

2.7.2 Basis Geometric Shapes

They are contained in the header file `basisElsh.h`. We make a distinction between `Basis Reference Shapes` (*BasRefSha*) and `Basis Geometric Shapes` (*GeoShape*). The firsts represent the basic geometries in the reference space. For instance, the class `Tetra`:

```
class Tetra
{
 public:
  static const ReferenceShapes Shape=TETRA;
  static const UInt nDim=3;
  static const UInt numVertices=4;
  static const UInt numFaces=4;
  static  const UInt numEdges=numFaces+numVertices-2;
};
```

These classes contains **only static quantities**. In particular

`Shape`	an `enum` variable used to identify the shape.
`nDim`	the dimension of the geometric figure identified by the class
`numVertices`	Number of vertices
`numFaces`	Number of faces
`numEdges`	Number of edges

At the time being we are effectively supporting only shapes whose faces are of the same type. The class `Prism` has not yet been considered, but its support is planned in the future. Figure 2.1 shows the numbering of vertices, edges and faces for the Basis Reference Shapes.

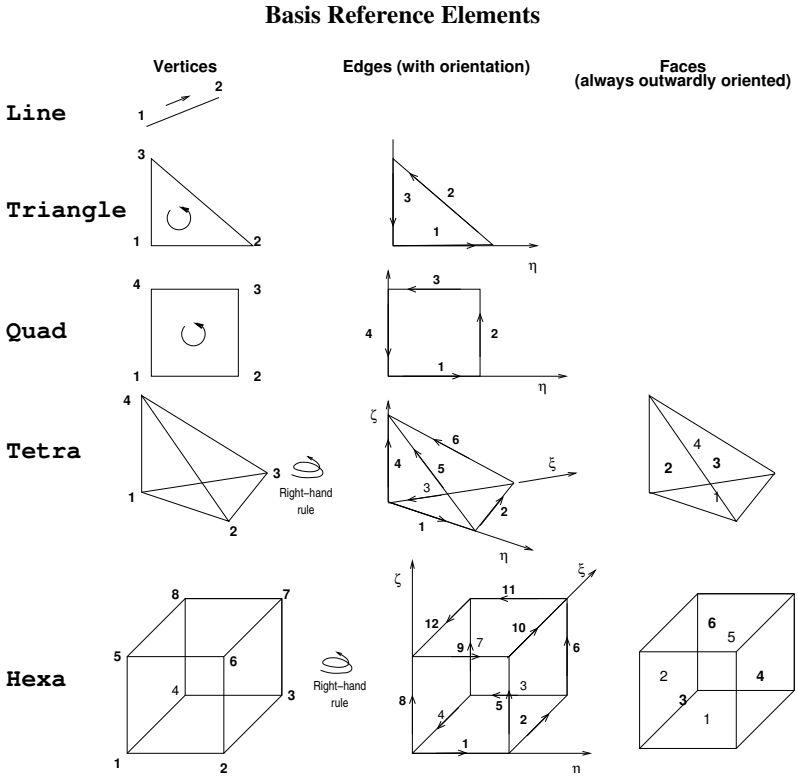

Figure 2.1: Numbering of vertices, edges and faces for the Basis Reference Shapes

A **Basis Geometry Shapes** ia a class derived from a *BasRefSha*. It is indeed a specialisation which adds to a *BasRefSha* additional basic information related to the `Points` which will define the shape of the actual geometrical entity. Again, here we include only the basic information, leaving to a more specialised class the knowledge of the coordinates of the points. An example of *GeoShape* is

```
class QuadraticTetra:
```

```
public Tetra
{
public:
  typedef Tetra BasRefSha;
  typedef QuadraticTriangle GeoBShape;
  static const UInt numPoints=10;
  static const UInt  numPointsPerEdge    = 1;
  static const UInt  numPointsPerFace    = 0;
  static const UInt  numPointsPerVolume  = 0;
  static UInt edgeToPoint(UInt const _localEdge, UInt const _point);
   static UInt faceToPoint(UInt const _localFace, UInt const _point);
};
```

Again all data are public and static, due to the basic nature of those classes (which are in fact `structs`).

`BasRefSha`	We export the type of the base *BasRefSha*.
`GeoBShape`	We export the *GeoShape* of the boundary.
`numPoints`	Number of Points
`numPointsPerEdge`	Number of Points **internal** to an edge
`numPointsPerFace`	Number of Points **internal** to a face
`numPointsPerVolume`	Number of Points **internal** to the volume (i.e. to the tetrahedron itself).
`edgeToPoint(i,j)`	j-th point on the i-the edge (numbering from 1)
`faceToPoint(i,j)`	j-th point on the i-the face (numbering from 1)

The quantity `numPointsPerVertex` is not stored because always equal to 1. We may note that other pieces of information are immediately deduced. If we take the example of the Quadratic Tetra we have that the total number of point on each face is `GeoBShape::`$S_numPointswhilethatfortheEdgesisgivenby$ **Numbering starts from 1**. Thus, if I want to know the maximum numbering of points on the edges of a `QuadraticHexa` I compute

`QuadraticHexa::`$S_numVertices + QuadraticHexa :: S_numPointsPerEdge * QuadraticHexa :: S_numEdges$`Quadratic`

2.7.3 Mesh Classes

The file `mesh_handler.h` contains all the remaining information relative to a `RegionMesh` entities. We start from the Geometric Elements

Geometric Elements

The geometric elements are the entities which stores the actual geometrical information relative to a `RegionMesh`. As usual, they are build by successive specialisation of small modules. The first are the `GeoD` and `GoeND` classes. The firsts stores basic information about the actual mesh points. The second is the base for the build-up of the other, 1d, 2D and 3D geometric entities.

```
class MeshVertex
{
 public:

  MeshVertex();
  MeshVertex(MeshVertex const & G);
  MeshVertex & operator=(MeshVertex const & G);

  Real const * coor() const {return _coor;};
  Real const & x() const;
  Real const & y() const;
```

```
  Real const & z() const;
  Real const & coordinate(UInt const i) const;

  UInt const & id() const { return _id;};
  bool const & boundary() const {return _boundary;};
  void showMe(bool verbose) const;

  UInt & id() { return _id;};
  Real & x();
  Real & y();
  Real & z();
  Real * coor() {return _coor;};
  bool & boundary()  {return _boundary;};
  Real & coordinate(UInt const i) ;
};
```

We give an explanation only of the methods whose meaning is not obvious.

id()	The unique identifier for the Point, numbering from 1. It corresponds to the position in the list of Points in `RegionMesh`. We have both the const and the non-const version. I need to find a way to hide the non-const version to the general user. Unfortunately the `template` arguments limit the use of the `friend` operator.
boundary()	True if boundary Point.
showMe(bool verbose)	Throw some info on the standard output. Should be rewritten better.

```
template <typename GeoShape>
class MeshElement {
 public:

  MeshElement();
  MeshElement(const MeshElement<GeoShape> &);
  MeshElement & operator=(MeshElement const & G);

  static const UInt numLocalPoints=GeoShape::S_numPoints;
  static const UInt numLocalVertices=GeoShape::S_numVertices;

  MeshVertex & point(UInt const i);
  MeshVertex const & point (UInt const i) const;

  // UInt const nPoints() const { return GeoShape::S_numPoints; };
  // UInt const nVertices() const { return GeoShape::S_numVertices;};

  UInt const & id() const { return _id;};

  void showMe(bool verbose) const;

  void setPoint(UInt const identity, point_Type  const & point); //put point
  bool setPointWithBoundaryCheck( ID const identity, point_Type const & point ); //with forced bound

  UInt & id() { return _id;};

};
```

The meaning of the variables is quite obvious. `MeshElement` template class has a `GeoShape` as template parameter, since some of its attribute depend from the geometry of the mesh we are

using.

The `MeshVertex` and `MeshElement` classes are never instantiated as such. They are building blocks for the actual `MeshElementMarked`n`D` template classes ($n = 1, 2, 3$), which add attributes more specific to a finite element mesh. The instance of a `MeshElementMarked`n`D` template class is what we call a *Geometry Element*, which we generally indicate as `GeoEle`.

Here the public interfaces

```
template <typename MARKER=PMarker>
class
MeshElementMarked0D: public MeshVertex, public MarkerHandler<MARKER>
{
 public:
MeshElementMarked0D();
MeshElementMarked0D(MeshElementMarked0D const & g);
MeshElementMarked0D(MeshVertex const & g, MARKER const & m);
MeshElementMarked0D & operator = (MeshElementMarked0D const & g);
};

template
<typename GeoShape, typename MARKER=EMarker>
class MeshElementMarked1D : public MeshElement<GeoShape>, public MarkerHandler<MARKER>
{
public:
  MeshElementMarked1D();
};

template
<typename GeoShape, typename MARKER=FMarker>
class MeshElementMarked2D : public MeshElement<GeoShape>, public MarkerHandler<MARKER>
{
public:

  MeshElementMarked2D();

  static const UInt numLocalEdges=GeoShape::S_numVertices;

  typedef typename GeoShape::GeoBShape EdgeShape;
  typedef  MeshElementMarked1D<EdgeShape> EdgeType;
  /* In the 3D case, we store the ids of the adjacent 3Delements.
     NOT THE POINTERS because we don't know the 3Delements type and
     I don't want to complicate the template declaration */
   ID firstAdjacentElementIdentity() { return M_firstAdjacentElementIdentity;}
   ID secondAdjacentElementIdentity(){ return M_secondAdjacentElementIdentity;}
   ID& firstAdjacentElementIdentity() { return M_firstAdjacentElementIdentity;}
   ID& secondAdjacentElementIdentity(){ return M_secondAdjacentElementIdentity;}
};

template
<typename GeoShape, typename MARKER=VMarker>
class MeshElementMarked3D : public MeshElement<GeoShape>, public MarkerHandler<MARKER>
{
public:

  MeshElementMarked3D();
```

```
  typedef typename GeoShape::GeoBShape FaceShape;
  typedef typename FaceShape::GeoBShape EdgeShape;
  typedef  MeshElementMarked1D<EdgeShape> EdgeType;
  typedef  MeshElementMarked2D<FaceShape> FaceType;

  static  const UInt numLocalFaces=GeoShape::S_numFaces;
  static  const UInt numLocalEdges=numLocalFaces+GeoShape::S_numVertices-2;
};
```

The name of the variables and methods should be self explanatory. Note the use of the `DEFINE` C Preprocessor variables `TWODIM` and `THREEDIM` used to identify portion of code specific to 2D and 3D problems, respectively. Again, we make an extensive use of constant `static` variables for attributes common to all class members.

The type of the geometry entity at the boundary is recovered through the `GeoBShape` data stored in the `GeoShape` class. We need to use the `typename` keyword to tell the compiler that the template attributes members are indeed types.

We use the Euler formula to determine some additional information, such as the number of Edges local to a 3D geometric Element.

The `MARKER` is a special class used to identify user defined attributes, such as boundary conditions. It is a still a rapidly evolving feature, which will be better detailed later on.

RegionMesh

Finally, the public interface of a `RegionMesh` class, which may be used to hold a 3D mesh. Here we make large use of *Standard Template Library* containers (and algorithms). This is probably the most complex class so far, at least for the part of the library concerning Mesh and Geometry handling. Lets look at its public interface piece by piece

```
template <typename GeoShape,
typename MC=MarkerCommon_Base> // Here we will add the Marker_Common class (TO DO)
class RegionMesh : protected MarkerHandler<typename MC::regionMarker>
```

This is a class template. The second template argument is particular and its discussion is postponed until te layout of the Marker classes is more or less at steady state. We mention only that it contains the typedefs of all the `Markers` for all the geometric elements (the one which have been defaulted to `PMarker`, `EMarker` etc. in the definition of the `MeshElementMarked` classes. Those types are exported to the `RegionMesh`, as usual, in order to be visible at class level without the need of applying the scope operator on the template argument:

```
public:

  typedef typename MC::pointMarker PointMarker;
  typedef typename MC::edgeMarker EdgeMarker;
  typedef typename MC::faceMarker FaceMarker;
  typedef typename MC::volumeMarker VolumeMarker;
  typedef typename MC::regionMarker RegionMarker;
```

Note By Luca We have now the mesh readers, i.e. the tools to read a mesh form a file. Here we have an architectural problem which has not been yet solved. We had three possibilities

1. The mesh reader is an external function passed as argument to the constructor;
2. The mesh reader is an external function which initialise an empty RegionMEsh object;
3. The mesh reader is a member function.

I have chosen the 3rd alternative, only as a matter of convenience. In fact, being a member function the reader may directly access the private members of the class. Of course, this is not a good thing, and since I took care that all the private member of the RegionMesh can be indeed accessed by a method of the class, it may be possible to rewrite the RegionMesh completely as an external module, which is more safe in case a user wants to add a new mesh reader. I will avoid option 1 since, although elegant, is a bit messy. •

```
bool readMppFile(const string & filename, UInt id, RegionMarker m);
```

A reader of files in `mesh++` format. Now a lot of typedefs which export to RegionMesh space the types used for the internal data.

```
typedef typename GeoShape::GeoBShape FaceShape;
typedef typename FaceShape::GeoBShape EdgeShape;

typedef  MeshElementMarked3D<GeoShape,VolumeMarker> VolumeType;
typedef  MeshElementMarked2D<FaceShape,FaceMarker>  FaceType;
typedef  MeshElementMarked1D<EdgeShape,EdgeMarker>  EdgeType;
typedef  MeshElementMarked0D<PointMarker>           point_Type;

// Typedefs for STL compliant containers of mesh geometric entities
// I Use VectorSimple for addressing from 1.

typedef VectorSimple<point_Type>    Points;  // Point List
typedef VectorSimple<VolumeType >  Volumes; // 3DElements list
typedef VectorSimple<FaceType> Faces;       /* Face list:Boundary Faces compulsory,
                                             if needed all faces. */
typedef VectorSimple<EdgeType> Edges;       /* Edges list:
                                             Filled only if needed, may be empty */
```

We use the `VectorSimple<T>` template class which is a simple wrap-around to the STL `vector<T>` template class (see header file `VectorSimpleor.h`) to store the list of the various Geometrical entities which form the Region Mesh. We follow the following Paradigm

A region mesh stores all the MeshElementMarkeds (Volumes in 3D problems, Faces in 2D problems etc), all the Points and all the Boundary MeshElementMarkeds. For all the other entities (for instance edges in a 3D problem) the storage is done according to a switch which is passed to the mesh reader. Details are still been worked out.

Now we have some methods which return some basic mesh data

```
UInt numLocalFaces() const // Number of local faces for each Volume
UInt numLocalEdges() const  // Number of local edges for each Volume
UInt numLocalEdgesOfFace() const //Number of edges on each face
UInt numElements() const // Total Number of Elements
UInt numVolumes() const // Total Number of Volumes
UInt numVertices() const //Total Number of Vertices
UInt numBVertices() const //Total Number of Boundary Vertices
UInt numPoints() const //Total Number of Points
UInt numBPoints() const //Total Number of Boundary Points
UInt numFaces() const //Total Number of Faces
UInt numBFaces() const //Total Number of Boundary Faces
UInt numEdges() const  //Total Number of Edges
UInt numBEdges() const //Total Number of Boundary Edges
```

We recall that `Volume` is the name of a 3D geometry element. Since `RegionMesh` handles only 3D problems, we also use the generic term `Element` to indicate, in fact, the volumes. That is we have `numElements()=numVolumes()`.

Also RegionMeshes have an `id`:

```
UInt const id() const; // Returns id

void showMe(bool verbose); // Prints some mesh info: must be done better!
```

Now all the methods which allow to access mesh geometric entities

```
point_Type const & point(UInt const i) const; // ith mesh point/vertex
FaceType const & face(UInt const i) const; // ith mesh face
EdgeType const & edge(UInt const i) const; // ith mesh edges
VolumeType const & volume(UInt const i) const; //ith mesh 3Delement
point_Type const & boundaryPoint(UInt const i) const; // ith b. point/vertex
FaceType  const & boundaryFace(UInt const i) const; // ith b. face.
EdgeType  const & boundaryEdge(UInt const i) const; // ith b. edge
```

One must be aware that some of the structures may be only partially filled up, or even completely empty. It depends on the data we are actually storing (which depends on the switches which have been set at mesh construction time (a feature still evolving!) To help the user, however, here there are some useful methods:

```
bool hasFaces() const;  // Do I store mesh faces?
bool hasInternalFaces() const; // Do I store also internal faces?
bool hasEdges() const;         // Do I store edges?
bool hasInternalEdges() const; // Do I store also internal edges?
```

Now we recall another **Paradigm**: points are numbered starting from the Vertices, the other entities starting from the one on the boundary. Yet, we have provided the following methods to interrogate geometry items (please note that they can take a reference to the object or its `id`)

```
bool isVertex(point_Type const & p) const;  //Is this point a Vertex?
bool isBoundaryPoint(point_Type const & p) const;  //Is this point on boundary?
bool isBoundaryFacet(FaceType const & f) const;  //Is this face on boundary?
bool isBoundaryEdge(EdgeType const & e) const;  //Is this edge on boundary?
bool isVertex(UInt const & id) const;  //Is this id a Vertex id?
bool isBoundaryPoint(UInt const & id) const;  //Is the Point with id on bdry?
bool isBoundaryFacet(UInt const & id) const;  //Same for a Face id
bool isBoundaryEdge(UInt const & id) const;  //Same for an Edge id
```

We now have some stuff which is not set up by default, for memory reason. For instance, the list of local faces etc. Here are the methods used to build up and interrogate those structures

We have first a method to get the elements adjacent to a Face. The method should works if the relative `Faces` object have been set up. Therefore, it always works for boundary faces, but not necessarily for internal faces (which are not set up by default).

The method returns the ID of the 3DElement adjacent to a Face. `Pos` is an `UInt` which may be equal to 1 or 2 and it indicates first or second Element. The first element is the one *ORIENTED coherently with the face AS STORED in Faces*. It means that the face orientation is OUTWARD with respect to the element. The second element is either null (boundary face) or indicates the element where the face appears INWARD oriented.

```
UInt  faceElement(UInt const faceId, UInt const Pos) const;
UInt  faceElement(FaceType const & f, UInt const Pos) const;
```

Now we have some other structures which are not set-up by default. This are the structures which give the global numbering of an Element local Faces or Edges. These structures are indeed used for the build up of the degrees of freedom, but, since they are not always necessary (for esample, for the setup of the degrees of freedom fro al linear Tetra we don't need them) we have avoided creating them by default (also because they eat up a lot of memory). The actual lists are kept hidden (private data), the user may only access them through the appropriate *constant* methods.

Note by Luca: Beware of the fact that the list of local faces/edges returns the id of a `BareFace` *(or* `BareEdge`*) (section 2.7.4). That means that an id number is return even if the actual Full* `Face` *or* `Edge` *has* ***not*** *been instantiated and it stored in the list of Faces and Edges of the mesh. This because one may want to have a numbering for Faces and Edges (for instance in order to build the Degrees of Freedom) without actually creating the corresponding Full Face or Edge object (i.e. the* `MeshElementMarked2D` *and* `MeshElementMarked1D` *object)! Therefore, if the user is interested in the Full Face of Edge object, he/she must first check whether it exists. Again, some helper methods are provided to make life a little easier.*

```
bool hasLocalFaces() const; // Is the local Faces array set up?
bool hasLocalEdges() const; // Same for Edges
```

There two functions are used to test whether the structures have been set up. If not, one may do:

```
void updateElementEdges(); // Build localEdgeId table
void updateElementFaces(); // Build localFaceId table
```

Now the methods which return the global nubering of local faces and edges on a volume element.

```
UInt localFaceId(UInt const volId, UInt const locF) const;
UInt localEdgeId(UInt const volId, UInt const locE) const;
UInt localFaceId(const VolumeType & iv, UInt const locF) const;
UInt localEdgeId(const VolumeType & iv, UInt const locE) const;
```

Now we may wish to know if a given id correspond to a full edge or face, that is if the corresponding `Edge` and `Face` objects have been instantiated.

```
bool isFullFace(UInt const & id) const;  //Full data for this face?
bool isFullEdge(UInt const & id) const;  //Full data for this edge?
```

2.7.4 Bare Items

The declarations and definitions of the `BareItem`s handlers are in `bareitem.h`. Since MeshElementBare are used only internally, we are (temporarily) omitting a detailed documentation. I (Luca) will only give an explanation of the genesis of the MeshElementBare. The RegionMEsh as it has been defined is able to store Edge and Face objects. Those objects are rather complex: they contains a Marker, possibly pointers to other structures etc. In other words, they use memory and one would like to avoid to instantiate them if not strictly necessary. Indeed, the RegionMesh has all the list of edges and faces ready, but in fact, only the list of boundary faces is filled by default (and probably one may avoid also that, if needed).

One of the paradigms chosen for the development of this library is the fact that degrees of freedom are linked to geometrical entities. Now if we have degrees of freedom associated, for instance, to Edges (like in a P2 Tetra) in order to build the global numbering of the Dof and the association between local (element wise) and global numbering I need to identify edges and give them an id number. Yet, maybe I don't want to build teh Edge object: after all all I need is the numbering and a way of getting the id's of the point on the edge, all the remaining data of the proper Edge object is not necessarily needed. (Beware, that is not anymore true if I want to associate to some edge associated Dof a boundary condition different from that of the two adjacent points!. In such case, you need to have a full edge!).

The dilemma has been resolved by creating the concept of bare edge and bare face. A bare item is formed by the minimal information required to uniquely identify it, namely 2 `Point`'s `id` for an edge and 3 `Point`'s `id` for the Faces (it is enough also for Quad faces!). We build the bare items by looping through the elements and obviously we make sure that the BareItem `id` is consistent with that of the corresponding "full item" if the latter has been instantiated. There is a little problem with the orientation of a BareItem on the boundary and the orientation of the corresponding full entity, which has not yet been fully resolved, but will be done soon.

2.7.5 BC Condition Classes

The file `bccond.h` contains the definition of the classes which are meant to hold boundary condition data. As a matter of fact, the use of this set of classes may be more general, they may be used, for instance, to hold domain based data and methods, such the viscosity function for a fluid flow computation, for instance. However, in the following we will only refer to the use of these classes for storing boundary condition data (and methods). Their extension to other uses is immediate. We will geneally refer to them as `BC` classes (and, correpsondingly, `BC` objects).

The classes are *estensible*, that is the user may add, by using the inheritance mechanism, specialised classes derived from the classes defined in the file. The mechanism here followed to introduce estensibility is that of *dynamic polymorphism*. By the use of dynamic polymorphism we have easily designed a container class which is able to uniformly store the various instances of `BC` Classes.

The major classes defined in the files are:

- A container class, called `BC_Handler`. The class is used to store and retrieve the various instances of `BC` Classes. It is based on the STL `set` container and it effectively stores pointer to `BC` objects, so it is able to provide a common uniform interface for base and user defined classes.
- Two base `BC` classes, called respectively `BC_Base` and `BC_Base_WL`. The default and user defined classes will be derived from them. Infact, `BC_Base_WL` in inherited from `BC_Base` and adds to it a vector of unsigned integers. The use of such a vector is to possibly store the list of Id's of the items to which that boundary condition is associated.
- Two default `BC` classes, called `SimpleBC` and `simpleBC_WL`, which are simply inherited form the corresponding base classes.
- A set of utilities to compare `BC` objects. Some of this utilities make use of the RTTI (run time type identification) technique.

Before giving more details on the `BC` classes, we introduce some nomenclature which will be used consistently in the following paragraphs.

type A boundary condition *type* refer to the capabilities of the boundary condition. Different types will correspond to *different classes* (all inherited from a `BC_Base`, or `BC_Base_WL` basis class). The user will create a new type only if he wants to enucleate some data and methods specific to a particular boundary treatment.

name The *name* of a boundary condition. It can be used to find a particular boundary conditions (if different names are used for different boundary conditions.

We now detail the various classes.

BC_Base and BC_Base_WL

```
class BC_Base
{
public:
```

```
  virtual void showMe()const;
  template <typename T>
  T& asLeaf(T const _t);
  template <typename T>
  T& asLeaf(T  * _t);
  virtual ~BC_Base(){}
  BC_Base(bcName_Type const _n);
#ifndef INT_BCNAME
  BC_Base(char const * _n);
#endif
  BC_Base();
  bcName_Type & name();
  bcName_Type const & name()const;
  bool unset()const;
protected:
  bcName_Type _name;
};
```

The class is very simple. It provides methods for explicit dynamic polymorphism (maybe they are useless). `showMe()` may be used for debugging purposes and `name()` may be used to set the name. The name may be also given through the constructor. The method `unset()` reset the name to the `nullBCname` value.

The `BC_Base_WL` is just an extension of the `BC_Base` class, with the addition of a STL container of unsigned integers.

```
class
BC_Base_WL: public BC_Base
{
public:
  BC_Base_WL(bcName_Type const _n):BC_Base(_n){};
#ifndef  INT_BCNAME
  BC_Base_WL(char const * _n):BC_Base(_n){};
#endif
  BC_Base_WL():BC_Base(){};
  virtual ~BC_Base_WL(){};
  virtual void showMe()const;
  vector<UInt> idList;
};
```

BC_Handler

The `BC_Handler` is a container of `BC` polymorphic objects. Indeed, it stores a STL set of pointers to `BC_Base` objects. It also provide a set of methods for adding, deleting an element from the set and for query the set. Furthermore the methods `size()` and `empty()` returns the result of the analog method applied to the set.

```
class BC_Handler
{
public:

  BC_Handler();

  typedef typename set<BC_Base *>::iterator   BC_PTR_iterator;
  inline BC_PTR_iterator end()const;
  inline BC_PTR_iterator begin()const;
```

```
  int size()const;
  inline bool  empty()const;
```

The BC_Handler methods which add/erase an element to the set; beware that the memory management routine must be done outside: the methods do not create/erase the BC object, they only operate on the container.

The meaning of method showMe() and of the various overloaded versions of isThere() is obvious. One may query the set either by giving a reference/pointer to a BC object, or by giving the name, since no objects with the same name may be present in the container.

```
  inline bool add(BC_Base  * const _bc);
  inline bool add(BC_Base & _bc);


  bool erase_BC(bcName_Type const  _n); /* Deletes entry with name _n

  inline bool  isThere(bcName_Type const _n) const;
  inline bool  isThere(BC_Base * const  _p) const;
  inline bool  isThere(BC_Base & _p) const;
  void showMe()const;
```

The method names() returns a STL vector with all the names in the container.

```
  vector<bcName_Type> & names()const;
```

Helper functions

There are a set of helper function, some of which are quite important. Two of those are the addBC and the addBC_ptr template function, which creates a new BC object and adds it to a BC_Handler. In principle, that function should have been an BC_Handler method, but a lack of full compliance with the treatment of template methods by the current version of *g++* compiler has made it necessary having *global functions*:

```
template
<typename BC=BC_Base>
BC &
addBC(BC_Handler & bh, bcName_Type const _n)

template
<typename BC=BC_Base>
BC *
addBC_ptr(BC_Handler & bh, bcName_Type const _n)
```

The functions CREATES a new BC object on the memory heap, add it to the container and return a reference or, respectively, a pointer to the created object.

The use of the function is immediate and illustarated in the following example, where both the reference and the pointer version are used.

```
// Some userdefined BCs. Just for fun!

class Dirichlet: public BC_Base
{
public:
  Dirichlet(bcName_Type _n): BC_Base(_n)
    {};
  static char* const type="Dirichlet";
```

```
  void showMe(){cout << type<< " " <<name()<<endl;}
};

class Neumann: public BC_Base_WL
{
public:
  Neumann(bcName_Type const  _n): BC_Base_WL(_n)
    {};
  static char* const type="Neumann";
  void showMe(){cout << type<< " " <<name()<<endl;}
};

int
main()
{
  BC_Base * pbc;
  BC_Handler Bound_cond;
  Dirichlet pippo=addBC<Dirichlet>(Bound_cond,"inflow");
  Dirichlet * pluto=addBC_ptr<Dirichlet>(Bound_cond,"wall");
  pluto->showMe();
  pippo.showMe();
  cout << "I am a "<< typeName(pippo)<<endl;
  cout << "I am a "<< typeName(pluto)<<endl;
  Neumann pw=addBC<Neumann>(Bound_cond,"wall");
  Bound_cond.erase_BC("wall");
}
```

In the previous example we have also shown some utilities which use the `RTTI` mechanism to test if two `BC` objects are of the same type, or to query their type. They are

```
inline bool sameType(BC_Base const * a, BC_Base const *  t)
inline bool sameType(BC_Base const & a, BC_Base const &  t)
inline bool sameType(BC_Base const & a, BC_Base const *  t)
inline bool sameType(BC_Base const * a, BC_Base const &  t)
inline char const *  typeName(BC_Base const & a)
inline char const *  typeName(BC_Base const * a)
```

Figure 2.2 shows a sketch of two `BC_Handler` objects storing two lists of `BC` objects.

2.7.6 Markers

The `Marker` classes are at the same time a "wrapper" around the `BC` class and the tool to be used to extend high level geometric classes with user defined function and methods. This latter aspect has been implemented by using the following technique. x

- A different marker class is defined for each 'high level' geometric element (Point, Line etc.). All those classes are in fact *extension* (by inheritance) of a basis class, called `Marker_Base`. `Marker_Base` just contains a pointer to a `BC` object and some methods to retrieve it. The user may create, **by inheritance from** `Marker_Base`, his own marker classes, where he will declare all data and methods which he wants to be available to the corresponding geometry element class. Indeed, the geometric element class *is derived* from the correponding Marker. Some basic marker classes are provided, to which the system defaults if the user does not provide his own.

- A class template, `MarkerCommon` takes as template parameter the typenames of Marker classes for all geometric elements. The `MarkerCommon` takes the defaults marker classes as

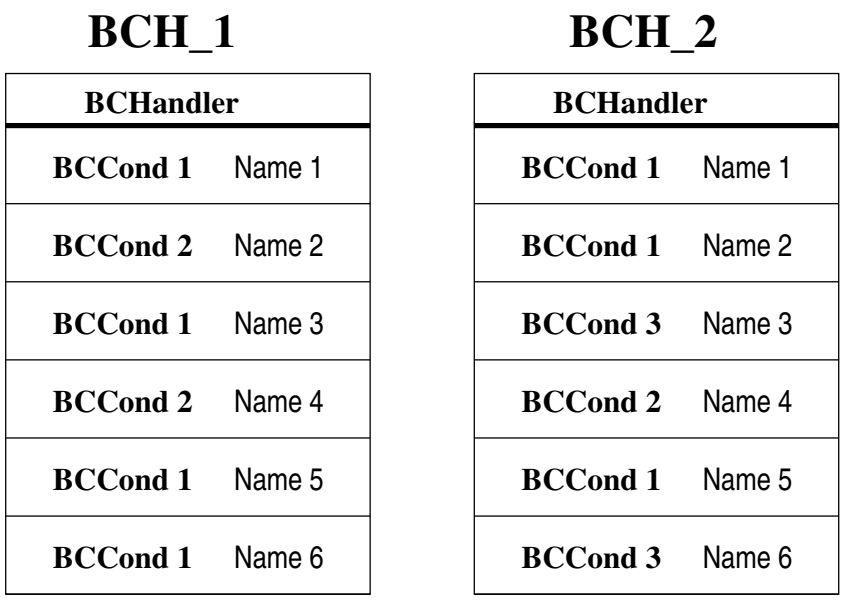

Figure 2.2: Sketch of two `BC_Handler` objects.

default values for the template arguments. The `MarkerCommon` is used as **mplate argument** for all high level geometry classes. This means, that a pecialised version, derived from the basis one, may be used to define *static* data and methods the user wants to be available to all high-level geometry entity classes.

In conclusion, extension to geometric entity classes is performed by inheritance from specialised Marker classes. Access to static data an methods to all geometric entity classes may be done by extending `MarkerCommon` (clearly, this is not the only way, global functions will work as well, but with less "data hiding").

In addition, a `MarkerHandler` class stores the container for the `BC` object. The default Marker for *Region* geometric entities, (RegionMarker_Base), id derived from the `MarkerHandler` class. This means, that in the default configuration we have one `BC` container at the level of a Region, while the other entities just store a pointer to a `BC` object. The user may change that if necessary.

As usual, some snapshots from the `markers_base.h` and `markers.h`

`markers_base.h`

Here the `Marker_Base` class which stores the `BC` pointer and provides the methods to extract it. The user `MUST BE WARNED` that in principle a `Marker_Base`, and thus the geometry item which derives from it, may not have a "boundary condition" associated. In order to avoid further complexities, this fact would be accounted by having a `NULL` pointer stored in the `Marker_Base` (and thus in the corresponding geometric entity). Therefore, some care MUST be taken when addressing the `BC` pointer. That is why the `MarkerUnset()` method has been provided.

```
#include "bccond.h"
class Marker
{
public:
  Marker():_marker(0);
  Marker(BC_Base * const _m ):_marker(_m);
```

```
  BC_Base * & marker_ptr();
  BC_Base * const & marker_ptr()const;
  bool markerUnset();
  BC_Base const & marker()const;
  template <typename BC>
  void
  setMarker(BC & _a);
  template <typename BC>
  void
  setMarker(BC  * const  _a);
protected:
  BC_Base * _marker;
};
```

The `MarkerHandler_Base` is just a wrapper for the `BC` container.

```
class MarkerHandler_Base
{
public:
  MarkerHandler_Base(){};
  BC_Handler & BCList(); \\ returns the list
  BC_Handler const & BCList();
protected:
  BC_Handler _lBC;
};
```

Here the default marker classes.

```
class DefPointMarker: public Marker
{
public:
  DefPointMarker(){};
  DefPointMarker(BC_Base * const _m ):Marker(_m){};
};

class DefFaceMarker: public Marker
{
public:
  DefFaceMarker(){};
  DefFaceMarker(BC_Base * const _m ):Marker(_m){};
};

class DefEdgeMarker: public Marker
{
public:
  DefEdgeMarker(){};
  DeEdgeMarker(BC_Base * const _m ):Marker(_m){};
};

class DefVolumeMarker: public Marker
{
public:
   DefVolumeMarker(){};
   DefVolumeMarker(BC_Base * const _m ):Marker(_m){};
};
```

The *default region marker* id derived also from `MarkerHandler_Base` class. *In the defaukt layout it is the region which has the list of boundary conditions.*

```
class DefRegionMarker:
public Marker, public MarkerHandler_Base
{
public:
   DefRegionMarker{};
   DefRegionMarker(BC_Base * const _m ):Marker(_m){};
};
```

The `MarkerCommon` class is used to store the typedefs of the user defined markers. If the user wants to create his own `MarkerCommon` (in order to add data and methods available to all geometric entities) he has to make a derivation from this one. The `MarkerCommon` templates arguments default to the default Marker classes.

```
template
<
class PM=DefPointMarker,
class EM=DefEdgeMarker,
class FM=DefFaceMarker,
class VM=DefVolumeMarker,
class RM=DefRegionMarker
>
class MarkerCommon
{
public:
typedef PM PointMarker;
typedef EM EdgeMarker;
typedef FM FaceMarker;
typedef VM VolumeMarker;
typedef RM RegionMarker;
};
```

We now define the `NULLMARKER` as the marker containing an empty pointer to `BC` object. It is useful for testing. An helper function is provided to test equality of markers (provisional, may be changed).

```
const Marker NULLMARKER=* new Marker;
inline bool operator==(const Marker & a, const Marker & b);
```

`markers.h`

This file just contains the type definition of the default `MarkerCommon`

```
#ifndef HH_MARKERS_HH_
#define HH_MARKERS_HH_
#include "Marker.h"
typedef MarkerCommon<
DefPointMarker,
DefEdgeMarker,
DefFaceMarker,
DefVolumeMarker,
DefRegionMarker
> DefMarkerCommon;
```

An example

Here we comment an example of usage of the default marker classes.

Some user defined `BC` classes, just for fun!

```
#include<iostream>
#include "mesh_handler.h"
class Dirichlet: public BC_Base
{
public:
  Dirichlet(string _n): BC_Base(_n)
    {};
  static char* const type="Dirichlet";
  void showMe(){cout << type<< " " <<name()<<endl;}
};

class Neumann: public BC_Base_WL
{
public:
  Neumann(string const  _n): BC_Base_WL(_n)
    {};
  static char* const type="Neumann";
  void showMe(){cout << type<< " " <<name()<<endl;}
};
```

Now we declare some geometric entities. Since we use only one template (the compulsory one which indicates the element basis shape) it means that we are using the default marker classes.

```
  MeshElementMarked1D<LinearLine> line;
  MeshElementMarked2D<LinearTriangle> tri;
  MeshElementMarked2D<LinearTriangle> tri1;
  MeshElementMarked2D<LinearTriangle> tri2;
  MeshElementMarked3D<LinearTetra> tet;
  RegionMesh<LinearTetra> mesh;
  MeshElementMarked0D<> point;
```

we now define a reference to the boundary condition handler, which (since we are using the default layout) is an attribute of the RegionMesh:

```
BC_Handler & BCl=mesh.BCList();
```

Now some examples of different ways of creating/assigning a boundary condition marker and storing/retrieving it to/from the list

```
  line.setMarker(addBC<Dirichlet>(BCl,"inflow"));
  tri.marker_ptr()=addBC_ptr<Dirichlet>(BCl,"inflow");
  tri1.marker_ptr()=addBC_ptr<Neumann>(BCl,"outflow");
  tri2.marker_ptr()=addBC_ptr<Neumann>(BCl,"outflow");
```

2.7.7 The pattern classes

The `pattern.h` file contains the definitions of the classes which are able to held the **pattern** of a global finite lements sparse matrix. A set of classes for pattern of **block matrices**, such as the Stokes matrix are also provided. The pattern class serves two purposes

1. Be of interface between the mesh and degree of freedom classes (sse section 2.7.14), the mesh and a (possibly external) linear algebra package whcih will directly use some of the pattern data for the build up of the actual matrices and the realted operations;

2. Be directly used for having some information about the *connectivity* of DOF (degree of freedom) data.

Some important convention

Since the pattern classes may act as an interface with an external linear algebra package, some attention has been paid in having the necessary flexibility both in the types holding the data that may be directly exchanged with the external package, and on the range of indices stored in the pattern. We have then introduced the following nomenclature and `typedefs`.

- **Raw pattern data** With this terminology we indicate the internal data stored in the pattern that may be passed *directly* to a linear algebra routine of an external package. For instance, the vectors that holds the row and column data (`ia` and `ja` in the notation by Saad [**?**]).
- **Index type**. The index type, identified by the typename `Index_t`, is the type of the raw pattern data. It defaults to `size_t` (stl integral type). It may be changed by the uset through the cpp macro `-DINDEX_T=`*`type`*. It **must be an integral type**, typical values are `int` or `unsigned int`.
- **Pattern Offset**. Depending wether we interface our solver with a C or a FORTRAN based algebra package, we may wish that the raw pattern indices start from 0 or 1. In order to maintain flexibility a cpp macro, *PATTERN_OFFEST* has been introduced. It defauults to `)`, yet it may be changed by compiling with `-DPATTERN_OFFSET=`*`num`*, being *num* either 0 or 1. The value of `PATTERN_OFFEST` is then transferred to the global constant variable `const Index_t PatternOffeset`.
- **Differences**. An difference type `Diff_t` is a type that holds an offset to a container. It is an integral variable which typically indicates the distance from the top of an array. Then its range is **always** starting from 1. It may be used to addredd arrays using the operator `[]`.

We recall that **identifiers** instead, start **always** from 1 and are indicated by the typename `ID`. The use of different typenames may help the programmer at identyfing the correct range of function arguments.

Main Pattern Classes

Here we give the list of the major pattern classes.

`class PatternDefs`	Base class for all patterns. It exposes basic types and some utility functions which are used in the implementation of the derived classes.
`typedef INDEX_T Index_t;`	Type for indices
`typedef vector<Index_t> Container;`	Container for the raw patterns
`typedef Container::size_type Diff_t;`	type for differences
`Index_t _d2i(ID const d) const;`	Converts an identifiers to the correpsonding index, depending on the value pf `PATTERN_OFFSET`
`ID _i2d(Index_t const i) const;`	From index to Identifier
`Diff_t _i2o(Index_t const i) const;`	From index to offset (difference)
`Diff_t _d2o(ID const d) const;`	From Identifier to offset (difference)
`Container & _i2d(Container & list_of_indices)const;`	Version which works on an entire container.
`Container & _d2i(Container & list_of_dof)const;`	Version which works on an entire container.
`Container & _d2o(Container & list_of_dof)const;`	Version which works on an entire container.

`class BasePattern`	`public PatternDefs`. Base class for simple patterns. It contains the common functionalities of all patterns (a parte mixed patterns used for block matrices).
`UInt nRows() const;`	Number of rows in pattern
`UInt nCols() const;`	Number of columns in pattern
`UInt nNz() const;`	Total on non-zero entries.
`bool isEmpty() const;`	True if pattern is empty.
`bool diagFirst()const;`	True if the raw pattern data contains the diagonal term as first entry.

```
class CSRPatt

CSRPatt();

CSRPatt(UInt nnz, UInt nrow, UInt ncol);

CSRPatt(UInt nnz, UInt nrow, UInt ncol, const vector<Index_t> &ia, const vector<Index_t> &ja );
```

TODO: TO BE COMPLETED

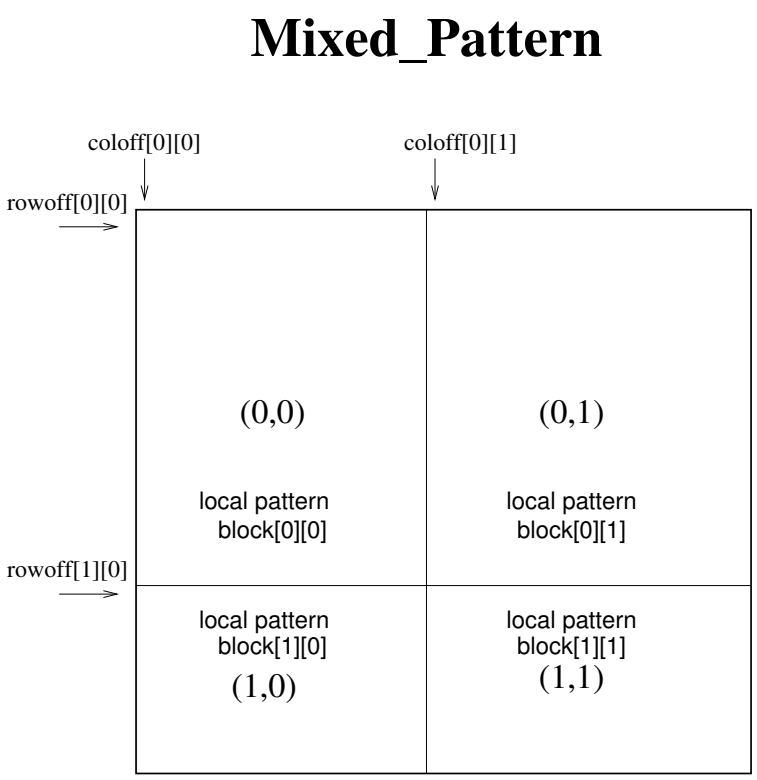

Figure 2.3: Schematic view of a 2×2 Mixed pattern

2.7.8 The Basis Function class

The `BasisFct` classes furnish the basis function for a finite element and for a mapping and are defined in `basisFct.h`. The basic finite elements and the related node numbering are illustrated in figure 2.4.

A `BasisFct` is basically a container of pointers to functions, which define the shape function and their derivatives. At the time being, only first derivatives have been considered, yet it may in principle be possible to add more in the future. The implementation is rather simple (and a little tedious) We start by defining a pointer to function type, to avoid exessive typing later on:

```
typedef const Real & cRRef;
typedef Real (* Fct)(cRRef,cRRef ,cRRef);
```

In order to simplify things we always use three arguments, even if the function is the shape function of a 2D element (or even a 1D one). Following a common procedure we start by defining a common interface which will then be specialised by deriving the actual classes It is a class template whose template arguments are a BasRefSha (see section 2.7.2) and an integer which indicates the number of shape functions. The use of a template argument for the number of shape functions allow to instantiate at compile time a vector of pointers to function. In this way we have a more efficient code and also a simpler constructor (indeed we use just the standard constuctor).

```
template<typename BRS,UInt _nbFct>
```

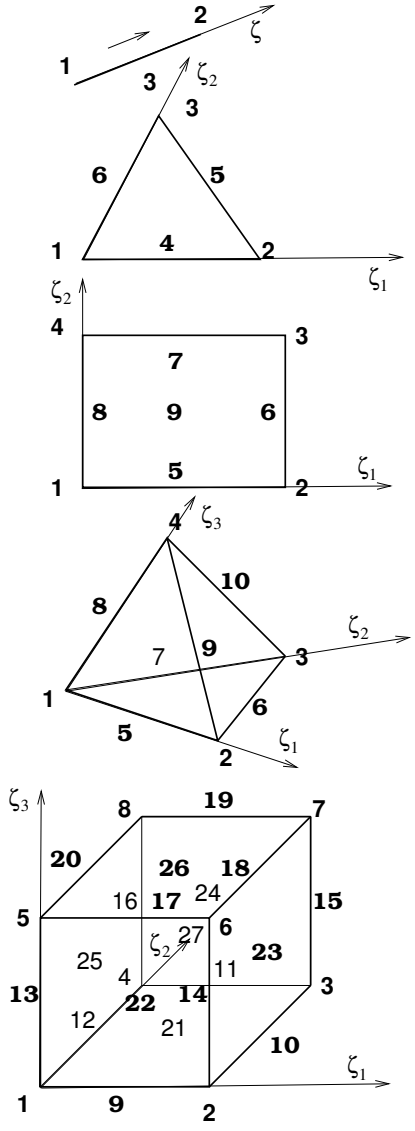

Figure 2.4: Finite element node numbering, both for linear and quadratic elements. Clearly, the lienar elements use only a subset of teh nodes here indicated. We use the general rule which imposes that the numbering follows the order: Vertices, Edges, Faces, Volumes (refer to figure 2.1)

```
class BasisFct_Base
{
public:
  static const UInt nbCoor = BRSE::S_nDimensions;/* For Compatibility
                                                    the name will be made uniform */
  static const UInt nbFct  = _nbFct; // Numero of Shape Functions
  typedef BRS BasRefSha; // Basic Reference Shape.
  Real fct(const UInt i,cRRef x,cRRef y, cRRef z) const;
  Real fctDer(const UInt i,const UInt icoor,cRRef x,cRRef y,cRRef z) const;
};
```

`nbCoor`	The number of coordinates. This variable is kept for compatibility only, since now this datum is available through the BasRefSha, and will be taken away soon.
`nbFct`	Number of Shape Functions. The template parameter is made visible.
`BasRefSha`	The Basic Reference Shape type. The template parameter is made visible.
`fct`	value of the i-th shape function at *reference coordinates* (x, y, z));
`fctDer`	*icoor* component $(1, 2$ or $3)$ of the i-th shape function derivative at *reference coordinates* (x, y, z)).

The rest of the file is the public interface for the various `BasisFct` classes, which implements the various finite elements. The details of the shape function definition may be found in the header file, here we report only the class declarations (the ones implemented so far). We signal here that the assigment of the function pointers to the correct shape functions (which are defined as global functions), is made by the class constructor. It is possible to define the functions without resorting to a global declaration, but it does not give any practical advantage (if not avoiding name clashes).

```
*/
class BasisFct_Q1_2D:     // Q1
  public BasisFct_Base<Quad,4>
{

};
class BasisFct_Q2_2D:     // Q2
  public BasisFct_Base<Quad,9>
{

};

class BasisFct_P1_3D:     // P1
  public BasisFct_Base<Tetra,4>
{

};

class BasisFct_P2_3D:     // P2
  public BasisFct_Base<Tetra,10>
{

};
```

Clearly the list is still short, but it will be incrased with time.

2.7.9 The geomap classes

Another building block for the finite element class is the mapping. The `geoMap.h` header file contains a prototype of the mapping classes.

This set of classes define the mapping between the actual and the reference finite element. In particular, for domain elements they should provide the methods for computing

- The correpondance $\boldsymbol{\zeta} \to \mathbf{x}$ between reference and actual coordinates;
- The jacobian matrix of the mapping and its determinant.

For a boundary element the classes should provide

- The correpondance $\hat{\boldsymbol{\zeta}} \to \mathbf{x}$ between reference and actual coordinates;
- The derivatives of the mapping $\frac{\partial \mathbf{x}}{\boldsymbol{\zeta}}$;
- The normal $\hat{\boldsymbol{\zeta}} \to \mathbf{n}(\boldsymbol{\zeta})$;
- The ratio between elementary surface areas.

Luca Note: There is still some work to do in order to provide a full set of estensible mapping classes, may be non necessarily based on a finite element type mapping (details may be found in the comments inside the header file). Also the internal organisation should be changed to make the classes more flexible. Furthermore, there is the need to define the mapping for a boundary element. Anyway, we here present the classes public interface as it is at the moment, which is already a very good and powerful starting point. •

A `GeoMap` is a class template with arguments a quadrature rule (`QuadRule`) a basis function (`BasisFct`) and a geometry element (`GeoEle`). The reason why we specify the quadrature rule is the the GeoMap needs to store some array with the value of the shape function and derivatives (provided by the `BasisFct`) at quadrature point, in order to have them ready all the time. The `GeoEle` is the geometry element (see section 2.7.3) on which a `Geomap` object is updated. The idea is in fact that there is only one instance of `Geomap` which is updated by passing the geometry data of a specific element.

```
template<class QuadRule,class BasisFct,class GeoEle>
class GeoMap
/*
|   #Prerequisites
|   BasisFct:nbFct must be less than equal Geoele::S_numPoints
*/
{
public:
  GeoMap();
  typedef          GeoEle      typeGeoEle;
  typedef          BasisFct    typeBasisFct;
  typedef          QuadRule    typeQuadRule;
  static const UInt nbMapNodes= BasisFct::nbFct;
  static const UInt nbCoor    = BasisFct::nbCoor;
  static const UInt nbQuadPt  = QuadRule::nbQuadPt;
```

As usual, some template argument type and variables are brought at the level of the classes, to avoid using name scope operators.

```
  void update(const GeoEle& geoele);
  void coorMap(Real& x,Real& y,Real& z,
               const Real & xi,const Real & eta, const Real &
               zeta) const; //(x,y,z) = F(xi,eta,zeta)
  void calcDetJac(Real det[nbQuadPt]);
  void calcDetInvJac(Real tInvJac[nbCoor][nbCoor][nbQuadPt],Real det[nbQuadPt]);
};
```

`update`	It takes a geometric element constant reference and updates he mapping on that element.It means that the successive calls to the class methods use the geometrical information of that element.**The user is warned that calling one of the other methods of the class without having *update*d the `GeoMap` object gives unpredictable results**.
`coorMap`	The map.
`calcDetJac`	Returns the jacobian determinant at quadrature points. It uses and array.;
`calcDetInvJac`	Inverse jacobian and its determinant at quadrature points. *Luca Note: I think that we should avoid using multidimensional native C++ arrays, because they are very error prone. We should give a closer look at the valarray classes of the STL.*

Now some global typedefs are created to make life easier.

```
typedef
GeoMap<QuadRule_Tetra_4pt,BasisFct_P1_3D,MeshElementMarked3D<LinearTetra> >
GeoMap_Linear_Tetra_4pt;
typedef
GeoMap<QuadRule_Tetra_1pt,BasisFct_P1_3D,MeshElementMarked3D<LinearTetra> >
GeoMap_Linear_Tetra_1pt;
typedef
GeoMap<QuadExact,BasisFct_P1_3D,MeshElementMarked3D<LinearTetra> >
GeoMap_Linear_Tetra_Exact;
typedef
GeoMap<QuadRule_Tetra_4pt,BasisFct_P2_3D,MeshElementMarked3D<QuadraticTetra> >
GeoMap_Quadratic_Tetra_4pt;
typedef
GeoMap<QuadRule_Tetra_1pt,BasisFct_P2_3D,MeshElementMarked3D<QuadraticTetra> >
GeoMap_Quadratic_Tetra_1pt;
```

The name of the types follws the pattern

`GeoMap_`*Fetype*`_`*n*`pt`,

where *Fetype* is the finite element type, such as `Linear_Tetra` and n is the number of quadrature points. The `QuadRule` template arguments (such as `QuadRule_Tetra_4pt`) are types defined in `quadRule.h` (see 2.7.10).

The type `GeoMap_Linear_Tetra_Exact` is used to specialise the `GeoMap` in case of exact integration (*Not by Luca: this part is still to be done*).

2.7.10 Quadrature Rules

This template class contained in the header file `quadRule.h`, provide the data for numerical integration. It has two template parameter, a `BasRefSha` (Basis Reference Shape, see section 2.7.2), and an unsigned integer which defines the number of quadrature points. It is a template parameter to allow static dimensioning of the arrays storing the weight and quadrature points (to enhance efficiency). The actual implementation of the class is done by *specialised constructors*[**?**]. Here teh class public interface:

```
template<typename BasRefSha, unsigned int _nbQuadPt>
class QuadRule
{
public:
  QuadRule();
  static const UInt nbQuadPt = _nbQuadPt;
  static const UInt nbCoor   = BasRefShaE::S_nDimensions;
```

```
  // For sake of simplicity it returns 0 if we are asking a coordinate
  // outide the scope of the Basis REference Shape
  // (for example the z coordinate for a triangle)
  Real const & quadPtCoor(UInt const ig,UInt const icoor) const;
  // It returns a pointer to an array containing the quad point
  // coordinate
  Real const * quadPtCoor(UInt const ig) const;
  Real const & weight(UInt const ig) const;
};
```

`nbQuadPt`	Number of quadrature points. As usual template arguments are brought to surface and made public.
`nbCoor`	The dimension of the geometric figure identified by the `BasRefSha`.
Real const `quadPtCoor`	*icoor* coordinate of the *ig*-th quadrature point (starting from 1). For sake of simplicity it returns 0 if we are asking a coordinate outiside the range $[1, \texttt{nbCoor}]$ (for example the 3^{rd} coordinate for a triangle)
Real const * `quadPtCoor`	Overloaded version which returns the whole quadrature point array for point `ig`.
`weight`	The *ig*-th point weight.

We have defined a **dummy** class template as a partial specialisation of the general class template, to be used for exact integration. The idea is to keep the class notation consistent and implement exact integration by template specialisation.

```
template<typename BasRefSha>
class QuadRule<BasRefSha,0>
{
public:
  static const UInt nbQuadPt = 0;
  static const UInt nbCoor   = BasRefShaE::S_nDimensions;
  Real const & quadPtCoor(UInt const ig,UInt const icoor) const
    {return nbQuadPt;};
  Real const * quadPtCoor(UInt const ig) const {return 0;};
  Real const & weight(UInt const ig) const {return 0;};
};
```

As usual, we use type definitions to save the user to remember all template parameters.

```
typedef QuadRule<Quad,1> QuadRule_Quad_1pt;
typedef QuadRule<Quad,4> QuadRule_Quad_2pt;
typedef  QuadRule<Quad,9> QuadRule_Quad_3pt;
typedef QuadRule<Tetra,0> QuadRule_Tetra_Exact;
typedef QuadRule<Tetra,1> QuadRule_Tetra_1pt;
typedef QuadRule<Tetra,4> QuadRule_Tetra_4pt;
```

Their meaning is evident.

2.7.11 Finite Element Classes

The actual finite elements classes are build it three step.

1. In the first step we define the *Base Finite Element* classes, which are concrete classes (no template) which contains the basis information about the degrees of freedom of a class of finite elements. In particular it contains the indication of the association of the elemental degrees of freedom with the geometrical items (see Paradigms, section 2.3) and the *local matrix pattern*. Their name ends in `_Base`, to indicate that they are still *Base* finite elements, i.e. the data there contained still refer to the reference element.

2. The `FeDef` class template, is an intermediate class template where a `GeoMap`, a Base Finite Element and a `quadRule` are put together to form an actual *finite element definition*. The reason why this is not the ultimate finite element class is related to the fact that we wish to support mixed finite elements. An efficient handling of mixed finite element within the framework chosen for the library requires to define this intermediate class.

3. The `FiniteElement` class, which completes all the data of a `FeDef` with the tools for computing the elemental matrices.

The `feDef.h` header file contains the Base Finite Element and the FeDef classes. Here an example taken from the header file:

```
class FE_P1_Tetra_Base
{
 public:
  typedef BasisFct_P1_3D BasisFct; // The Shape Functions.
  //
  static const UInt  nbNode       = BasisFct::nbFct;
  static const UInt  nbCoor       = BasisFct::nbCoor;
  //
  // Dof Information
  //
  static const UInt  nbDofPerVertex  = 1;
  static const UInt  nbDofPerEdge    = 0;
  static const UInt  nbDofPerFace    = 0;
  static const UInt  nbDofPerVolume  = 0;
  //
  //Pattern Information
  //
  static const UInt  nbPattern    = nbNode*nbNode;
  static const UInt  nbDiag       = nbNode;
  static const UInt  nbUpper      = nbNode*(nbNode-1)/2;
  //
  static UInt patternFirst(const UInt i);
  static UInt patternSecond(const UInt i);
};
```

`BasisFct`	Generic typename which indicates the Basis Shape Function class used by the finite element.
`nbCoor`	The dimension of the associated geometrical element (taken from the `BasisFct`)
`nDofPrrVertex`	The number of degrees of freedom attributed to a `Vertex` of the associated geometrical element. **Beware:** the number of degrees of freedom associated to a geometry entity is related to the order of the finite element, and are invariant to the fact that we are treating a scalar or a vector problem. In other words, a linear triangle will always have one one degree of freedom per vertices, also when used to solve a vector problem. Moreover, we are here talking of `Vertices`, **not Points!**.
`nbDofPerEdge`	The number of degrees of freedom attributed to an `Edge` of the associated geometrical element. Beware that the Dof associated to the Edge ends are indeed DofPerVertex!
`nbDofPerFace`	The number of degrees of freedom attributed to an `Face` of the associated geometrical element.
`nbDofPerVolume`	The number of degrees of freedom attributed to the `Volume` geometrical element. Beware: this are the number of Dof "internal" to the volume, **not** the total number of Dof of the element.
`nbPattern`	Number of elements of the local matrix, usually equal to `nbNode*nbNode` (*Luca Note: Indeed we may think of a generic pattern class, to be used for the standard cases*);
`nbDiag`	Number of diagonal terms in the local matrix, Usually equal to `nbNode` (*See previous note*);
`nbUpper`	Number of terms in the upper triangula part of the local matrix;
`patternFirst`	First term of the i-th element of the pattern ;
`patternSecond`	Second term of the i-th element of the pattern.

Patterns are organised so that the first local matrix element stored are those on the diagonal, then the remaining terms are sorted (consequently the lower triangular terms are first than the upper triangular ones); The pattern data will be obviously used for the build-up of teh global matrices

We now give a closer look to the `feDef`:

```
template <typename GM, typename BFE>
class FEDef : public BFE
{
public:
  typedef  typename GM::typeQuadRule QuadRule;/* Quadrule by GeoMap,
                                          to ensure consistency:
                                          (TODO) to be done better! */
  typedef GM  GeoMap;
  typedef typename GeoMap::typeGeoEle   GeoEle;
  static const UInt  nbQuadPt     = QuadRule::nbQuadPt;
};
```

It may be noticed that it is a class template which puts together the basic ingredients of a finite element. The quadrule is not indicated becouse it is implicitely defined by the mapping (*Luca Note: This is something that will be changed later on*) Finally, the header file provides some types (only a few, a lot more will be added!)

FE_P1_Tetra_1pt	P1 Tetra finite element, 1 point of quadrature.
FE_P1_Tetra_4pt	P1 Tetra finite element, 4 point of quadrature.
FE_P1_Tetra_Exact	P1 Tetra finite element, Exact quadrature.It **MUST** be specialised, and that is done in the `fe_p1_3d.h` header file.
FE_P2_Tetra_1pt	P2 Tetra isoparametric finite element, 1 point of quadrature.
FE_P2_Tetra_4pt	P2 Tetra isoparametric finite element, 4 point of quadrature.
FE_P2_Linear_Tetra_1pt	P2 Tetra affine finite element, 1 point of quadrature.
FE_P2_Linear_Tetra_4pt	P2 Tetra affine finite element, 4 point of quadrature.

Finally, we look at the public interface of the template class in the `finiteEle.h` header file

```
enum DerivSwitch {Measure,FirstDeriv,SecondDeriv};

//======================================================================
//
//                        Finite Element
//
//======================================================================
template<class FE>
class FiniteEle:
  public FE
{
public:
  FiniteEle();
  //
  static const UInt nbDerivSwitch=3;
  bool getDerivSwitch(const DerivSwitch _s) const;
  UInt currentID() const;
  void globCoorQuadPt(Real& x,Real& y,Real& z,const UInt ig) const;
  void update(const typename FE::GeoEle & geo_ele,const DerivSwitch d_sw=FirstDeriv);

  Real measure();
  void mass(Real* mat,const Real coef) const;
  void stiff(Real* mat,const Real coef) const;

  void source(Real* vec,const Real constant) const;
  template<class UsrFct> void source(Real* vec,const UsrFct& fct) const;
};
```

This class template will be the basic block fo r**user defined finite element classes**. The `DerivSwitch` is a Switch (see header file `switches.h`) which is used to indicate the type of data at quadtrature points the finite element builds when it is `updated` on an actuale geometrical element.

2.7.12 The specialised versions for exact integration

2.7.13 The Mixed finite element classes

2.7.14 The Degrees of Freedom

2.7.15 The Fields

2.7.16 The assembly process

Chapter 3

HOWTO

3.1 Geometrical mappings

In file `geoMap.h`:
GeoMap : analytical mapping for parametric Lagrangian map
GeoMapQuadRule : derived from GeoMap when a quadrature rule is given
GeoMap is the base class, two kinds of classes can be derived from it:

- GeoMapQuadRule : when a quadrature rule is used
- various classes when an "exact" integration is used

3.2 HOWTO add new basis functions

The basis functions may be used to define the geometrical mappings and the finite element shape functions. A lot of basis functions are already implemented in the file `basisFct.h`. To create a new set of basis functions you must create a class which derives from the class `BasisFct_Base<Shape,nbFct>},`where Shape is either Line or Quad or Triangle or Tetra or Hexa, and nbFct is the number of basis functions. Your class has just to contain a constructor which fills the arrays fct[i] ($0 \leq i < nbFct$), and fctDer[i][icoor], ($0 \leq i < nbFct$ and $0 < icoor < Shape :: S_nDimensions$). These arrays contain pointers to the basis functions and their derivatives. These function are staightforwardly coded as global function which take the coordinates x, y, z as arguments and return the value of the function (see examples in the file `basisFct.h`).

3.3 HOWTO add a new quadrature rule

The way is very similar to the one described for the basis functions. You have just to derive a new class from the class `QuadRule<Shape,nbQuadPt>}`.See the file `quadRule.h`

3.4 HOWTO add a new element operator

By "element operator", we mean element matrix corresponding to a differential operator. For example, the element stiffness or mass matrices.

•**First case**: you use a quadrature rule. Just add your operator in the file `elemOper.h` by copying and modifying an existent one (e.g. stiff or mass). By this way, the new operator will work with arbitrary general finite elements.

•**Second case**: if you want to code "hardly" (without integration rule) the element matrix corresponding to your operator for a given finite element, you must create a "specialized" finite element. This may be much more efficient, but the new operator will work only with this element.

3.5 HOWTO build a new finite element

In `feBase.h`: create a base class for the new element (copy, paste and modify an existant one...), say `FE_My_Tetra_Base}`.The following depends on what you want to do.

•**First case**: you want to use a quadrature rule. The advantage: you will straighforwardly inherit all the work already done for the other finite elements (in particular the operators defined in `elemOper.h`). The drawback: if your finite element has a low order, the comptationnal cost may be more expensive than with a "specialized integration".

- have a look at the end of the file `geoMap.h`, you will find many geometrical mapping together with integration rules. Choose one, say `GeoMap_Linear_Tetra_4pt`. If you don't like the geometrical mappings already defined, you have to create a new one. Remember that a geomapQR class is the "tensorial product" (template in fact...) of a set of basis function (in this example, `BasisFct_P1_3D`, see `basisFct.h`), a geometric element (here, MeshElementMarked3D¡LinearTetra¿, see `basisElSh.h`) and a quadrature rule (here, `QuadRule_Tetra_4pt`, see `quadRule.h`).

- in `finiteEleQR.h`: create a typedef correponding to the "product" of your base finite element and the geomap:

```
typedef FiniteEleQR<FE_My_Tetra_Base,GeoMap_Linear_Tetra_4pt> FE_My_Tetra_4pt;
```

•**Second case**: you don't want to use a quadrature rule. Advantage: probably more efficient. Drawback: you have to reimplement your own element operators. Create a `MyGeoMap` class where you can store some geometrical stuffs, and a class `MyFE` where you store what you want accordingly to you finite element. The job now consists in developping the *specialized* operators. It should look like

```
template<>
void stiff<MyFE,MyGeoMap>(Real coef,ElemMat& elmat,const MyFE& fe,
const MyGeoMap& geo,int iblock=0,int jblock=0)
...
```

Thanks to the specialization, as soon as you adopt the same interface for your operator than those of `elemOper.h`, a code that works with a "general" FE (i.e. with a quadrature rule) will work without any change with your new element.

You can have a look at `fe_p1_tetra_exact.h`. But I emphasize that this is just a test, and it should probably be improved (in particular, the geometrical quantities stored in the geomap class...).