ISTITUTO
DI
ANALISI NUMERICA
del
CONSIGLIO NAZIONALE DELLE RICERCHE
via Abbiategrasso 209 – 27100 PAVIA (Italy)

PAVIA
1999

PUBBLICAZIONI

N. 1164

Enrico Bertolazzi, Gianmarco Manzini

P2MESH: Programmer’s Manual

P2MESH: Programmer's Manual

Enrico Bertolazzi[1] & Gianmarco Manzini[2]

[1]*Department of Mechanics and Structures Engineering*
University of Trento
via Mesiano 77, I – 38050 Trento, Italy
Enrico.Bertolazzi@ing.unitn.it

[2]*Institute of Numerical Analysis – CNR*
via Ferrata 1, I – 27100 Pavia, Italy
Gianmarco.Manzini@ian.pv.cnr.it

Abstract

P2MESH was developed for the solution of partial differential equation in two dimensions on unstructured meshes. The library is a collection of C++ classes and iterators which allows to design and implement the data structures involved in Finite Element and Finite Volume methods. This report documents the methods in the public interface for all the library classes.

(NO) Installation

The `P2MESH` software library consists in the header file `p2mesh.hh` to be included at the beginning of each program source file using `P2MESH` facilities. **No installation** or pre-compilation of library files is required. No library object or archive files must be linked.

Conditions for Using p2mesh

Acknowledgements

We have a long list of people to thank for the interest they manifested about `P2MESH` and the encouragement they gave us. In alphabetical order we mention Dr. Mario Arioli, Dr. Antonio Cazzani, Dr. Loula Fezoui, Prof. Bruno Firmani, Dr. Luca Formaggia, Dr. Alessandro Russo, Prof. Gianni Sacchi, Prof. Filippo Trivellato, and Dr. Gianluigi Zanetti. Finally, we would like to address special thanks to Prof. Bruce Simpson, Dr. J.-Daniel Boissonat and all the team of the project Prisme at INRIA, Sophia-Antipolis, France, for the opportunity of the first official presentation of the work.

Contents

1 Preface

The P2MESH software package comprises five base classes acting as templates for the definition of data types in any user application. The package also encompasses a set of suitable iterators.

The design of a software application based on P2MESH basically consists in the specification of a suitable set of derived classes, that would embody both the physical and numerical details in the corresponding partially pre-defined “geometric” types provided by the library.

As a result, it is generally not useful to instantiate mesh-based objects directly from the library types, because these instances would show nothing more than their geometrical nature.

Throughout the manual, the base classes in P2MESH will also be referred as *library classes*, and the derived classes as *project classes*, being *the project* a generic user application based on P2MESH.

The presentation of a P2MESH data type is given in four parts which introduce the reader to the public interface. All the features and functionalities of the library classes are discussed, which may be inherited by public derivation. The four parts appear under the headers Class Name, Description, Usage, and Member Functions. Sometimes there are some *remarks*, to focus the reader attention on a particular issue.

The following table indicates the names of the P2MESH library classes, the conventional names adopted in the manual for the project classes, and the nature (geometrical or not) of the type.

Base Class Name	Derived Class Name	Container Type
`p2_common`	Common	shared information
`p2_vertex`	Vertex	vertex instance
`p2_edge`	Edge	edge instance
`p2_poly`	Poly	polygon instance
`p2_mesh`	Mesh	mesh instance

The prefix `p2_` in the base class names has been adopted in the library implementation in order to avoid name conflicts with other project names, that, for instance, might be defined by the user or which might be already defined in other software packages.

From the base class `p2_poly` two different types of polygons can be derived, triangles and quadrilaterald. The keyword Poly thus refers to a generic polygon type and any information concerning an instance of such a type will hold for both triangle and quadrilateral objects.

The instance of the project classes Vertex, Edge, Poly and Mesh will be simply called *vertices*, *edges*, *polygons* and *mesh*.

The attribute `const` is usually omitted in the public method declarations for the sake of compactness. Moreover the default values for the template argument `int`, `unsigned`, `double` are used throughout the manual in the method declarations. However, the complete form of the declarations can be easily found in the kernel description [1] or in the source code.

Note that in the Usage section, the presentation emphasizes the viewpoint of the project classes by discussing some simple source code fragments which are proposed as examples of typical and practical applications. The usage section can thus be used as a reference guide for the programmer using `P2MESH`.

2 p2_common public interface

Class Name `p2_common`

Description The class `p2_common` is a base container class for the public derivation of the class Common, which is, as suggested by its name, the user defined *common* class of the project. The class Common should only be a container for the static data and the `typedef`s alias definitions which are shared by the instances of the other project classes. No instances of the library class `p2_common` and of the project class Common should normally be instantiated in the code.

Internal Prototype The template header declaration of the class `p2_common` is

```
 1 template <typename P2V_type,
 2           typename P2E_type,
 3           typename P2P_type,
 4           typename P2M_type,
 5           unsigned SIZE_value    = 3,
 6           bool     LIST_value    = false,
 7           typename REAL_type     = double,
 8           typename INTEGER_type  = int,
 9           typename UNSIGNED_type = unsigned,
10           typename VMARK_type    = unsigned,
11           typename EMARK_type    = unsigned,
12           typename PMARK_type    = unsigned>
13 class p2_common ;
```

Lines 1–4 introduce the names of the project classes. Line 5 specifies the number of the vertices of each polygon (the default value assumes a triangular mesh). Line 6 introduces a boolean flag which allows the user to set the internal implementation of the vertex class. Lines 7–9 introduce the project numerical types for integer and real numbers, which may be either standard C++ built-in ones or other user-defined ones. This last alternative handles the case where a higher precision arithmetic is devised for the user application by means of numerical types from some specific software package. Lines 10–12 define the project type for markers. The default value is `unsigned` however it can be any built-in or user-defined type. In the case of the user defined type the user **must** provide the operators `>>` and `<<` in order to use markers with the `read_mesh` and `read_map_mesh` methods.

The following `typedef` are defined inside the class and are accessible in all the derived classes:

```
typedef P2V_type      P2V   ;
typedef P2E_type      P2E   ;
typedef P2P_type      P2P   ;
typedef P2M_type      P2M   ;

typedef VMARK_type    Vmark ;
typedef EMARK_type    Emark ;
typedef PMARK_type    Pmark ;

typedef REAL_type     Real ;
typedef INTEGER_type  Integer ;
typedef UNSIGNED_type Unsigned ;
```

Usage The library classes `p2_vertex`, `p2_edge`, `p2_poly` and `p2_mesh` are parametrized by the project class Common. Each project class is then publicly derived from the corresponding library class whose template header contains the project class Common.

A triangular mesh is specified by the following code fragment where only project class names are given as class template parameters and all other choices are given by default.

```
class Common : public p2_common<Vertex, Edge, Poly, Mesh> {
// private definitions
public:
// public definitions
} ;
```

A quadrilateral mesh, instead, is specified by explicitly introducing `SIZE=4` in the template header declaration.

```
class Common : public p2_common<Vertex, Edge, Poly, Mesh, 4> {
// private definitions
public:
// public definitions
} ;
```

Two different internal implementations of the vertex base class `p2_vertex` are supported by `P2MESH`. These implementations mainly differ in the explicit availability of the vertex connectivity, which may be present as lists of first neighbor vertices, incident edges and polygons and a set of public methods which return the related information. Since vertex connectivity lists are expensive to produce and demand a large amount of computer memory, `P2MESH` allows the user to decide by means of the boolean flag `List` whether the connectivity lists must be built during the initialization phase of the mesh-manager and stored in memory. The default value is `List=false` and no vertex connectivity list is available, while the choice `List=true` selects the other case, where all lists are built and stored. Hence, if the vertex connectivity of a triangular mesh is needed, the following code fragment must be used.

```
class Common : public p2_common<Vertex, Edge, Poly, Mesh, 3, true> {
// private definitions
public:
// public definitions
} ;
```

The standard built-in arithmetic types `double`, `int`, `unsigned` are parameterized by using the alias names `Real`, `Integer` and `Unsigned`. The alias names are accessible within the project classes; for the sake of clarity, throughout the manual we use their default values `double`, `int` and `unsigned`. The following code fragment shows how different numerical types can parametrize the internal implementation of the library. Floating-point real numbers are defined by the high-precision type `doubledouble`[1], and `long` and `unsigned long` are used instead of respectively `int` and `unsigned`.

```
class Common : public p2_common<Vertex, Edge, Poly, Mesh,
                                4, false,
                                doubledouble, long, unsigned long> {
// private definitions
public:
// public definitions
} ;
```

The marker types are also parameterized by using `Vmark` for vertex markers, `Emark` for edge markers and `Pmark` for polygon markers. If no user type is specified, the

[1]http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html

default type is `unsigned`. For example, the following code fragment shows how `double` markers can be used instead of `unsigned`.

```
class Common : public p2_common<Vertex, Edge, Poly, Mesh,
                                3, false,
                                double, int, unsigned,
                                double, double, double> {
// private definitions
public:
// public definitions
} ;
```

Notice that in this case you must specify all the template arguments.

Member Functions No member functions.

3 p2_vertex public interface

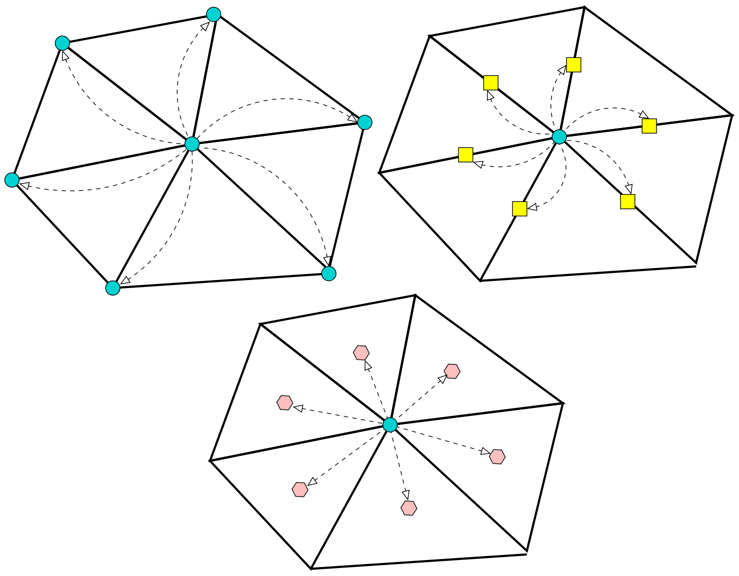

Figure 1: Vertex-Vertex, Vertex-Edge and Vertex-Polygon connections in the definition of a p2_vertex instance (pointers are optionally stored in memory)

Class Name p2_vertex

Description The class p2_vertex is the base class for the public derivation of the class Vertex, which is the user defined *vertex* type of the project. The private attributes are two floating point numbers for the coordinates of the vertex. Optionally, pointer lists of connected vertices, edges and polygons may be stored.

Usage The vertex class of the user application is denoted by Vertex, and is constructed by public derivation from the vertex class p2_vertex of the library. p2_vertex is parametrized by class Common, which is the common class defined by the user. The type Common must appear as argument in the template argument list of the library class p2_vertex

```
class Vertex : public p2_vertex<Common> {
// private definitions
public:
// public definitions
} ;
```

Member Functions The table shows the public methods which are available for any instance of type Vertex.

n.	Method	Description
1	`unsigned n_vertex(void)`	number of connected vertices
2	`unsigned n_edge  (void)`	number of incident edges
3	`unsigned n_poly  (void)`	number of incident polygons
4	`Vertex & vertex(unsigned i)`	reference to the `i`-th connected vertex
5	`Edge   & edge  (unsigned i)`	reference to the `i`-th incident edge
6	`Poly   & poly  (unsigned i)`	reference to the `i`-th incident polygon
7	`unsigned local_number(Vertex & v)`	local id of vertex `v`
8	`unsigned local_number(Edge   & e)`	local id of edge `e`
9	`unsigned local_number(Poly   & p)`	local id of polygon `p`

These member functions are available only if the user explicitly sets the option by selecting `List=true` in line `6` of the template header definition of the class `p2_common` on page 9. The member functions (1–3) return the number of the connected vertices, edges and polygons to the current vertex instance. The member functions (4–6) return references to the connected mesh entities. The member functions (7–9) return the position of the connected mesh entities inside the vertex lists. When the template parameter `List=false` in `p2_common`, methods (1–3) return the value `0` while methods (4–9) produce a run-time error.

There is no particular order in the elements returned by functions (4–6) the only constraint is that if `V` is a reference to a vertex, either the couple of vertices

`( V, V . vertex(i) )` or `(V . vertex(i), V)`

defines the edge

`V . edge(i)`

n.	Method	Description
10	`double & x()`	vertex first coordinate
11	`double & y()`	vertex second coordinate

The member functions (10–11) return the values of the coordinates of the current `p2_vertex` instance.

4 p2_edge public interface

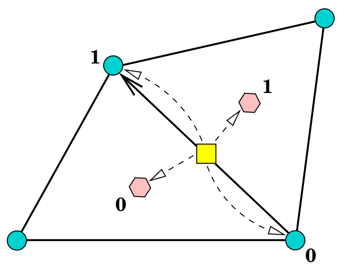

Figure 2: Edge-Vertex and Edge-Polygon connections for an edge in a triangle-based mesh.

Class Name `p2_edge`

Description The class `p2_edge` is the base class for the public derivation of the class Edge, which is the user defined *edge* type of the project. The private attributes are:

- the pointers to the two Vertex instances in the geometrical definition of the current edge;
- the pointers to the two adjacent Poly instances in the geometrical definition of the current edge; the second pointer is set to `NULL` if the edge is on the boundary.

Remark Each edge is oriented from the first vertex to the second one. The orientation uniquely defines a normal direction to the edge, conventionally oriented from the left to the right side of the edge. The pointers to the adjacent polygons are such that the first one always refers to the polygon on the *left* edge side. On boundary edges, the normal vector is always defined in the outward direction, that is the unique adjacent polygon is always located on the left side and the pointer to the right side polygon is automatically set to `NULL`.

Usage The edge class of the user application is denoted by Edge, and is constructed by public derivation from the edge class `p2_edge` of the library. `p2_edge` is parametrized by Common, which is the common class defined by the user. The type Common must appear as the argument in the template header specification of the library class `p2_edge`

```
class Edge : public p2_edge<Common> {
// private definitions
public:
// public definitions
};
```

Member Functions The following methods return the topological information stored as private attributes in the current `p2_edge` instance

n.	Method	Description
12	`unsigned n_vertex(void)`	return 2
13	`unsigned n_edge  (void)`	return the number of adjacent edges
14	`unsigned n_poly  (void)`	return the number of adjacent polygons
15	`Vertex & vertex(unsigned i)`	reference to vertex `i`
16	`Edge   & edge  (unsigned i)`	reference to edge `i`
17	`Poly   & poly  (unsigned i)`	reference to polygon `i`
18	`unsigned local_number(Vertex & v)`	local id of vertex `v`
19	`unsigned local_number(Edge   & e)`	local id of edge `e`
20	`unsigned local_number(Poly   & p)`	local id of polygon `p`
21	`bool     ok_poly(unsigned i)`	true if polygon `i` exists

Method (12) returns 2, that is the number of vertices which define the edge. Method (13) returns the number of adjacent edges. For a triangle-based mesh this number is 4 when the current edge is internal and 2 when it is located on the boundary. These numbers become respectively 6 and 3 for a mesh of quadrilaterals. Method (14) returns the number of adjacent polygons. This number is 2 when the current edge is internal and 1 when it is located on the boundary. Method (15) returns the reference to the first (second) vertex of the current edge when `i=0` (`i=1`). Method (16) returns the references to the edges on the left and right side for respectively

- Triangle-based mesh: left `i=0,1`, right `i=2,3`.
- Quadrilateral-based mesh: left `i=0,1,2`, right `i=3,4,5`.

Figure 3 depicts the local references to the adjacent edges. Method (17) returns the reference to the left (right) adjacent polygon when `i=0` (`i=1`). Methods (18-20) and (21)

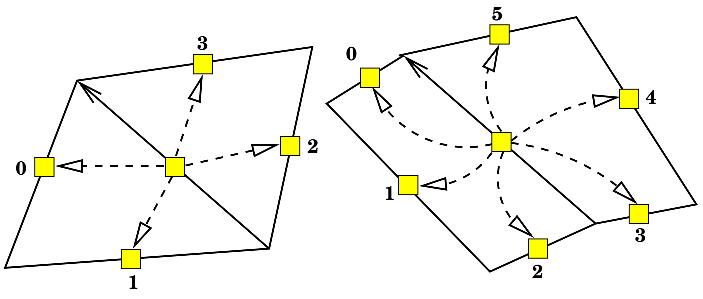

Figure 3: Surrounding edge numbering for triangles and quadrilaterals

are self explanatory. The following public methods return the geometrical information which are stored as private attributes in the current edge.

n.	Method	Description
22	`double & x(unsigned i)`	vertex `i`: first coordinate
23	`double & y(unsigned i)`	vertex `i`: second coordinate
24	`double xm()`	midpoint first coordinate
25	`double ym()`	midpoint second coordinate
26	`double xt(double & t)`	interpolated first coordinate
27	`double yt(double & t)`	interpolated second coordinate
28	`double nx()`	first component of normal vector
29	`double ny()`	second component of normal vector
30	`double tx()`	first component of tangent vector
31	`double ty()`	second component of tangent vector
32	`double length()`	edge length

Methods (22–23) are similar to the statements

`vertex(i).x()` and `vertex(i).y()`.

Notice that the value of the coordinates cannot be changed by the user application by means of the former methods.

Methods (24–25) return the coordinates of the edge midpoint, i.e.

$$\texttt{xm()} = \frac{x_0 + x_1}{2}$$
$$\texttt{ym()} = \frac{y_0 + y_1}{2}$$

Methods (26–27) return the coordinates of the linearly interpolated point $(x(t), y(t))$ on the edge at the location $t \in [0, 1]$. The interpolation is given by the affine mapping

$$x(t) = x_0 + t\,(x_1 - x_0)$$
$$y(t) = y_0 + t\,(y_1 - y_0)$$

Note, also, that `xt(0.5)` and `yt(0.5)` return the same values returned by `xm()` and `ym()`.

Methods (28–29) return the components of the vector $\mathbf{n}$ =`(y(1)-y(0),x(0)-x(1))`, which is orthogonal to the edge, oriented counterclockwise (as specified in the previous remark). Methods (30–31) return the components of the vector **t** =`(x(1)-x(0),y(1)-y(0))`, which is parallel to the edge, oriented counterclockwise. These vectors are not normalized, and their lengths are equal to the length of the edge which is returned by method (32).

Remark The information returned by methods (24–32) are not stored as part of the edge definition, but it is evaluated on the current edge instance each time the corresponding method is invoked. When the same information must be used several times, it is convenient to introduce those quantities in the definition of the project edge class. This strategy allows some CPU time saving at the price of a greater storage requirement. The following code fragment illustrates the issue.

```
class Edge : public p2_edge<Common> {
  Real stored_xm, stored_ym, stored_nx, stored_ny, stored_lenght;
  // additional private definitions
public:

  void SetUp(void) {
    stored_xm     = p2_edge<Common>::xm() ;
```

```
 7      stored_ym     = p2_edge<Common>::ym() ;
 8      stored_nx     = p2_edge<Common>::nx() ;
 9      stored_ny     = p2_edge<Common>::ny() ;
10      stored_length = p2_edge<Common>::length() ;
11    }

12    // overload library function
13    Real const & xm    (void) const { return stored_xm ; }
14    Real const & ym    (void) const { return stored_ym ; }
15    Real const & nx    (void) const { return stored_nx ; }
16    Real const & ny    (void) const { return stored_ny ; }
17    Real const & length(void) const { return stored_length ; }

18    // additional public definitions
19  }
```

The method `SetUp` assigns some computed geometrical quantities to the corresponding attributes defined in the project class `Edge`. The member functions in lines `13`–`17` override the ones inherited from the base class `p2_edge`. The method `SetUp` must be called by the application for *any* instance of type `Edge` at the initialization step. Note that in the method `SetUp`, the alias `Real` which internally parametrizes the library is used instead of the built-in arithmetic type `double`.

5 p2_poly public interface

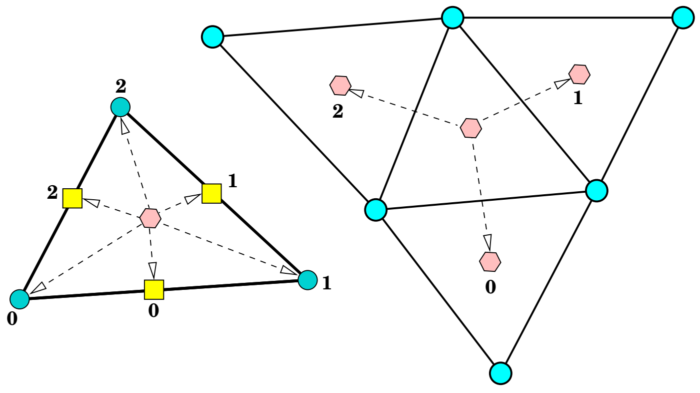

Figure 4: On the left Polygon-Edge and Polygon-Vertex connections for a triangular p2_poly instance (the pointers are stored in memory). On the right Polygon-Polygon connections (the pointers are determined at run-time)

Class Name p2_poly

Description p2_poly is the base class for the public derivation of the project class Poly which is the user defined polygon type of the project. The private attributes are

- the pointers to the n vertices in the definition of the current polygon

$$\left(\mathsf{v}^0, \mathsf{v}^1, \ldots, \mathsf{v}^{n-1}\right) ;$$

- the pointers to the n edges in the definition of the current polygon

$$\left(\mathsf{e}^0, \mathsf{e}^1, \ldots, \mathsf{e}^{n-1}\right) .$$

The number of vertices or edges in the polygon definition is by default $n = 3$, which corresponds to a triangle-based mesh, and can be set to $n = 4$ for a quadrilateral-based mesh.

Remark **(the edge orientation)**
The set of vertices listed in a polygon forms a closed path which is always oriented counterclockwise in the case of an external boundary and clockwise on an internal one. This fact requires that any edge in the definition of a polygon be oriented in a very precise way. Nevertheless, the orientation of an edge in the mesh is determined by the order in which its two vertices are stored in the edge data structure, which is independent of the way edges are memorized in polygons. Moreover, the two polygons sharing an internal edge would always "see" it with opposite orientation!

To solve this consistency problem, it is crucial to recognize the orientation of the edge. To this purpose, the public interface of the class `p2_poly` implements the method `ok_oriented()`, which returns the boolean `true/false` depending the orientation of an edge in the current polygon *is/is not* the same in edge definition.

We emphasize that the orientation of an edge specifies also the orientation of the normal vector to that edge. When an edge has the same orientation in the mesh data set and in the actual definition of a polygon instance, its normal vector is oriented outward that polygon. Remember also that a boundary edge is always built in such a way that its normal vector is oriented outward the computational domain. Therefore, boundary edges always form a counterclockwise closed path of an external boundary and a clockwise closed path of an internal one. A boundary edge must also show the same orientation in the mesh data set and in the definition of the unique polygon instance it belongs to.

Usage The polygon class of the user application is denoted by Poly, and is constructed by public derivation from the polygon class `p2_poly` of the library. `p2_poly` is parametrized by Common, which is the common class defined by the user. The type Common must appear as the argument in the template header list of the library class `p2_poly`

```
class Poly : public p2_poly<Common> {
// private definitions
public:
// public definitions
} ;
```

Member Functions

Topological methods

The following methods return the topological information stored as private attributes in a `p2_poly`-type object

n.	Method	Description
33	`unsigned n_vertex()`	number of vertices
34	`unsigned n_edge  ()`	number of edges
35	`unsigned n_poly  ()`	number of adjacent sides
36	`Vertex & vertex(unsigned i)`	reference to vertex `i`
37	`Edge   & edge  (unsigned i)`	reference to edge `i`
38	`Poly   & poly  (unsigned i)`	reference to polygon `i`
39	`unsigned local_number(Vertex & v)`	local id of vertex `v`
40	`unsigned local_number(Edge   & e)`	local id of edge `e`
41	`unsigned local_number(Poly   & p)`	local id of polygon `p`
42	`bool ok_poly    (unsigned i)`	check the existence of the `i`-th polygon
43	`bool ok_oriented(unsigned i)`	check the edge orientation

Methods (33–35) return the number of vertices, edges and adjacent sides of a given polygon. The three methods always return the same value, since the number of vertices, edges and adjacent sides necessarily coincide. However, it is clearer and safer to distinguish them when writing loops on vertices, edges, or adjacent polygons.

Methods (36–37) return the reference to the `i`-th vertex or edge in the polygon definition. Method (38) returns the reference to the polygon adjacent to the `i`-th edge. If this edge is located on the domain boundary, a run-time error is produced, because that polygon does not exist. This case is also checked by method (42).

Method (39) accepts an input reference to a vertex and returns the local id in the vertex list of the polygon. Method (40) accepts an input reference to a edge and returns the local id in the edge list of the polygon. Method (41) accepts an input reference to an adjacent polygon and returns the its local id. Whenever the input reference is not correct a run-time error is produced. Typical mistakes consist in referencing to a vertex or an edge not belonging to the current polygon, or to a polygon which is not adjacent.

Method (43) verifies whether the edge orientation in the polygon (which is always counterclockwise) is the same as the orientation of the edge instance in the mesh data set. Method (42) accepts in input the local identifier of the edge and returns `true/false` whether the edge `is not/is` located on the boundary.

Geometrical methods

The following public methods return some geometrical information.

n.	Method	Description
44	`double & x(unsigned i)`	vertex `i`: first coordinate
45	`double & y(unsigned i)`	vertex `i`: second coordinate
46	`double xm(unsigned i)`	edge `i`: midpoint first coordinate
47	`double ym(unsigned i)`	edge `i`: midpoint second coordinate
48	`double xt(unsigned i, double & t)`	edge `i`: interpolated first coordinate
49	`double yt(unsigned i, double & t)`	edge `i`: interpolated second coordinate
50	`double nx(unsigned i)`	edge `i`: normal vector first component
51	`double ny(unsigned i)`	edge `i`: normal vector second component
52	`double tx(unsigned i)`	edge `i`: tangent vector first component
53	`double ty(unsigned i)`	edge `i`: tangent vector second component
54	`double length(unsigned i)`	edge `i`: length
55	`double xc()`	centroid first coordinate
56	`double yc()`	centroid second coordinate
57	`double area()`	polygon area

Methods (44–45) are equivalent to `vertex(i).x()` and `vertex(i).y()` but the coordinate values cannot be changed.

Remark **(Midpoint and linearly interpolated points on a polygon edge)**
Methods (46–47) return the coordinates (x, y) of the midpoint of the `i`-th edge in the current polygon. The midpoint is, to some extent, a special one, because it is equidistant from both the edge vertices and its location is independent of the edge orientation. Thus, if `P` is a reference to a polygon, the statement

```
double x = P . xm(i) ;
```

is equivalent to the statement

```
double x = P . edge(i) . xm()
```

Methods (46–47) are equivalent to `edge(i).xm()` and `edge(i).ym()`.

Instead, methods (48–49) return the coordinates (x, y) of the linearly interpolated point at $t \in [0, 1]$ on the `i`-th edge. The point location is determined by the choice of the origin on the edge, i.e. by the edge orientation. Thus, if `P` is a reference to a polygon, the statement

```
  double xt = P . xt( i, 0.3 ) ;
```

is **not** equivalent to the statement

```
  double xt = P . edge(i) . xt( 0.3 ) ;
```

because the edge may **not** be oriented in the same way as in the current polygon instance. For the correct equivalence, refer to the following statement

```
double xt ;
if ( P . ok_oriented(i) )
  xt = P . edge(i) . xt( 0.333 ) ;
else
  xt = P . edge(i) . xt( 1 - 0.333 ) ;
```

which takes into account the edge orientation.

Remark The multiple accessing syntax is allowed. For example, if `pPoly` is a pointer to a polygon object, the coordinates of the centroid of the adjacent polygon `i` can be accessed by

```
  double xc = pPoly -> poly(i) . xc() ;
  double yc = pPoly -> poly(i) . yc() ;
```

The simultaneous usage of both the deferentiation operators "`->`" and "`.`" may appear rather cumbersome. Notice that it is just a matter of choosing in the library design whether methods (36-38) should return a reference or a pointer. In the authors' opinion, the second alternative is safer, since it forces the final user to work with references. Inelegant multiple accessing can be avoided by adopting expressions like

```
  Poly & P = pPoly -> poly(i) ;
  double xc = P . xc() ;
  double yc = P . yc() ;
```

which generally require one more statement but are still elegant and perhaps clearer.

Methods (50–51) are *not* equivalent to `edge(i).nx()` and `edge(i).ny()` because orientation depend on edge orientation. For example

```
double nx = P . nx(1) ;
```

is equivalent to

```
double nx ;
if ( P . ok_oriented(1) )
  nx = P . edge(i) . nx() ;
else
  nx = - P . edge(i) . nx() ;
```

Analogously methods (52–53) are *not* equivalent to `edge(i).tx()` and `edge(i).ty()`. For example

```
double tx = P . tx(1) ;
```

is equivalent to

```
double nx ;
if ( P . ok_oriented(1) )
  tx = P . edge(i) . tx() ;
else
  tx = - P . edge(i) . tx() ;
```

Remark Methods (38) and (46–57) return values which are not stored as private attributes in the actual instance, but are evaluated each time the corresponding method is invoked. This implementation choice, as in the analogous case for `p2_edge`, is motivated by the intention of limiting the memory required by the library classes. A project application which makes intensive and repeated usage of this kind of information may however result too expensive and therefore computationally inefficient. However, CPU costs can be reduced and computational efficiency can be improved by estimating these geometrical quantities only once and then storing them as attributes of the project classes. The following source fragment code illustrates the case.

```
 1 class Poly : public p2_poly<Common> {
 2   Real stored_xc, stored_yc, stored_area ;
 3   // additional private definitions
 4 public:
 5   void SetUp(void) {
 6     stored_xc   = p2_poly<Common>::xc()   ;
 7     stored_yc   = p2_poly<Common>::yc()   ;
 8     stored_area = p2_poly<Common>::area() ;
 9   }
10   // override library member functions
11   Real const & xc  (void) const { return stored_xc   ; }
12   Real const & yc  (void) const { return stored_yc   ; }
13   Real const & area(void) const { return stored_area ; }
14 }
```

The member functions in lines `11–13` override the homonymous ones inherited from the class `p2_poly`. The method `SetUp` must be called for any polygon instance during the initialization phase. Notice that the alias `Real`, whose definition is contained in the project common class (see page 10), is used instead of the numerical built-in type `double`.

5.1 Reference frame supporting functions

The `P2MESH` library provides the user application with a minimal set of suitable methods to map each polygon (either triangle or quadrilateral) to a reference polygon.

The triangle case

The reference triangle T_{ref} is the simplex defined in the (s,t) coordinate system by

$$\mathsf{T}_{ref} = \{(s,t) \mid s \geq 0,\ t \geq 0,\ s+t \leq 1\}.$$

Let T denote a generic triangle defined by the ordered list of vertices $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$, whose coordinates are indicated by

$$\mathbf{p}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}.$$

By introducing the two displacement vectors $\mathbf{u}$ and $\mathbf{v}$

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0 = \begin{pmatrix} u_x \\ u_y \end{pmatrix}, \qquad \mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0 = \begin{pmatrix} v_x \\ v_y \end{pmatrix},$$

the non-singular affine mapping from T_{ref} to T takes the form

$$\begin{cases} x(s,t) = su_x + tv_x + x_0 \\ y(s,t) = su_y + tv_y + y_0 \quad , \end{cases} \tag{1}$$

and the inverse mapping is

$$\begin{cases} s(x,y) = \dfrac{v_x(x-x_0) - v_y(y-y_0)}{u_y v_x - u_x v_y}, \\ t(x,y) = \dfrac{u_y(x-x_0) - u_x(y-y_0)}{u_y v_x - u_x v_y}. \end{cases} \tag{2}$$

The quadrilateral case

The reference quadrilateral Q_{ref} is the polygon defined in the (s,t) coordinate system by

$$\mathsf{Q}_{ref} = \{(s,t) \mid -1 \leq s \leq 1, -1 \leq t \leq 1\}.$$

Let Q denote a generic quadrilateral defined by the ordered list of vertices $\mathsf{Q} = (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$, whose coordinates are indicated by

$$\mathbf{p}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}.$$

By introducing the following displacement vectors

$$\mathbf{c} = \frac{\mathbf{p}_0 + \mathbf{p}_1 + \mathbf{p}_2 + \mathbf{p}_3}{4}, \qquad \mathbf{u} = \frac{-\mathbf{p}_0 + \mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3}{4},$$
$$\mathbf{v} = \frac{-\mathbf{p}_0 - \mathbf{p}_1 + \mathbf{p}_2 + \mathbf{p}_3}{4}, \qquad \mathbf{w} = \frac{\mathbf{p}_0 - \mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3}{4},$$

the non-singular affine mapping from Q_{ref} to Q takes the form:

$$\begin{cases} x(s,t) = su_x + tv_x + stw_x + c_x, \\ y(s,t) = su_y + tv_y + stw_y + c_y. \end{cases} \tag{3}$$

The inverse mapping is

$$
\begin{aligned}
s &= \frac{1}{a_s}\begin{cases} -b_s(x,y) + d_s(x,y) & \text{if } b_s d_s > 0 \\ -b_s(x,y) - d_s(x,y) & \text{otherwise} \end{cases}, \\
t &= \frac{1}{a_t}\begin{cases} -b_t(x,y) + d_t(x,y) & \text{if } b_t d_t > 0 \\ -b_t(x,y) - d_t(x,y) & \text{otherwise} \end{cases},
\end{aligned} \tag{4}
$$

where

$$
\begin{aligned}
\mathbf{c}_0(x,y) &= \mathbf{p}_0 + \mathbf{p}_1 - 2(x,y)^T, & \mathbf{d}_0(x,y) &= \mathbf{p}_0 + \mathbf{p}_3 - 2(x,y)^T, \\
\mathbf{c}_1(x,y) &= \mathbf{p}_2 + \mathbf{p}_3 - 2(x,y)^T, & \mathbf{d}_1(x,y) &= \mathbf{p}_1 + \mathbf{p}_2 - 2(x,y)^T, \\
\mathbf{c}_2 &= \mathbf{p}_1 - \mathbf{p}_0, & \mathbf{d}_2 &= \mathbf{p}_3 - \mathbf{p}_0, \\
\mathbf{c}_3 &= \mathbf{p}_2 - \mathbf{p}_3, & \mathbf{d}_3 &= \mathbf{p}_2 - \mathbf{p}_1,
\end{aligned}
$$

and

$$
\begin{aligned}
a_s &= \mathbf{c}_2 \wedge \mathbf{c}_3, \\
b_s(x,y) &= \mathbf{c}_2 \wedge \mathbf{c}_1(x,y) + \mathbf{c}_0(x,y) \wedge \mathbf{c}_3, \\
c_s(x,y) &= \mathbf{c}_0(x,y) \wedge \mathbf{c}_1(x,y), \\
d_s(x,y) &= \sqrt{b_s^2(x,y) - 4a_s c_s(x,y)},
\end{aligned}
$$

$$
\begin{aligned}
a_t &= \mathbf{d}_2 \wedge \mathbf{d}_3, \\
b_t(x,y) &= \mathbf{d}_2 \wedge \mathbf{d}_1(x,y) + \mathbf{d}_0(x,y) \wedge \mathbf{d}_3, \\
c_t(x,y) &= \mathbf{d}_0(x,y) \wedge \mathbf{d}_1(x,y), \\
d_t(x,y) &= \sqrt{b_t^2(x,y) - 4a_t c_t(x,y)},
\end{aligned}
$$

with

$$\mathbf{p} \wedge \mathbf{q} = p_x q_y - p_y q_x.$$

n.	Method	Description
58	`void st_to_xy(double & s, double & t double & x, double & y)`	affine mapping from reference to actual element
59	`void xy_to_st(double & x, double & y, double & s, double & t)`	affine mapping from actual to reference element
60	`void jacobian(double & s, double & t, double J[2][2])`	Jacobian of the mapping
61	`void inverse_jacobian (double & s, double & t, double InvJ[2][2])`	inverse Jacobian of the mapping

Methods (57–60) will refer to equations (1)–(2) for a triangle-based mesh or to equations (3)–(4) for a quadrilateral-based mesh.

6 p2_mesh public interface

Class Name `p2_mesh`

Description `p2_mesh` is the library class for the public derivation of the project class Mesh. It is the most complex data structure provided by the library. The private attributes include the lists of

- vertices:
$$\left(\mathsf{v}^0, \mathsf{v}^1, \ldots, \mathsf{v}^{n_v-1}\right) ;$$
- edges:
$$\left(\mathsf{e}^0, \mathsf{e}^1, \ldots, \mathsf{e}^{n_e-1}\right) ;$$
- polygons:
$$\left(\mathsf{p}^0, \mathsf{p}^1, \ldots, \mathsf{p}^{n_p-1}\right) ;$$

where n_v, n_e and n_p are respectively the total number of vertices, edges and polygons contained in the mesh.

Usage As is the case of the other data structure hitherto examined, the project class Mesh is publicly derived from the library class `p2_mesh` and parametrized with the common project class Common, which appears explicitly in the template argument list of the base class `p2_mesh`.

```
class Mesh : public p2_mesh<Common> {
  // private definitions
public:
  // public definitions
}
```

Member Functions The main geometrical and topological information are accessible by the following methods:

n.	Method	Description
62	`unsigned n_vertex ()`	number of vertices
63	`unsigned n_bvertex()`	number of boundary vertices
64	`unsigned n_ivertex()`	number of internal vertices
65	`unsigned n_edge ()`	number of edges
66	`unsigned n_bedge()`	number of boundary edges
67	`unsigned n_iedge()`	number of internal edges
68	`unsigned n_poly ()`	number of polygonal elements
69	`unsigned n_bpoly()`	number of boundary polygons
70	`unsigned n_ipoly()`	number of internal polygons
71	`Vertex & vertex(unsigned i)`	reference to vertex `i`
72	`Edge   & edge  (unsigned i)`	reference to edge `i`
73	`Poly   & poly  (unsigned i)`	reference to polygon `i`
74	`unsigned local_number(Vertex & v)`	local id of vertex `v`
75	`unsigned local_number(Edge   & e)`	local id of edge `e`
76	`unsigned local_number(Poly   & p)`	local id of polygon `p`
77	`void bbox(double & xmin, double & ymin, double & xmax, double & ymax)`	bounding box of the mesh

Methods (62–64), (65–67) and (68–70) return the total number of vertices, edges and polygons in the mesh and the number of internal and boundary objects.

Methods (71–73) return the reference to the `i`-th vertex/edge/polygon in the mesh data set. Methods (74–76) accept a reference to an instance of a given type and return its internal location, that is the location in the corresponding mesh data set container. The internal location may be used as the integer identifier (global number) of that instance. As usual in `C` and `C++` all numberings start from `0`.

Method (77) returns the coordinates of the bottom-left and the top-right points defining the bounding box of the mesh.

6.1 Mesh builders

The mesh data set is automatically initialized by invoking a mesh builder method. In the present implementation several different mesh builders are available, which perform quite different actions. The mesh builders can be logically grouped whether they start from a *structured* description of the computational mesh or they accept an *unstructured* list of coordinates and connectivities.

The former ones are

(78) `tensor_mesh`,

(79) `std_tensor_mesh`,

(80) `map_mesh`,

(81) `read_map_mesh`;

while the latter ones are

(82) `build_mesh`,

(83) `read_mesh`.

The methods (78–81) build a regular mesh of triangles – three different orientations are possible – or quadrilaterals. They start from a *structured* description of the domain triangulation, which is internally generated for the first three methods, and given as an input data file for the fourth one. The method `map_mesh` allows a re-mapping of the computational domain onto a generic shaped four-side domain by a user provided coordinate transformation function.

The method (82) initializes the mesh data set starting from an *unstructured* description of the initial triangulation, provided by the user application as lists of coordinates and connectivities while the method (83) performs the mesh data set initialization starting from a input data files in the output format of the mesh generator `Triangle`. For both methods, the list of vertex coordinates and the list of connections “polygon $\to$

vertices" are mandatory; a list of edges can be optionally given in input. The Edge-type data structures are initialized by using such a list when available, otherwise they are internally built.

The nature of the polygons to be generated by the mesh builders, that is triangles or quadrilaterals, is not explicitly indicated as an entry argument of the mesh builder methods. This information is already specified as an argument in the template header declaration of the project class Common.

Remark **(Markers)**
It is a common practice in programming numerical algorithms that a marker, usually an integer identifier, be assigned to mesh geometrical entities, such as vertices, edges or polygons. The marker is generally used to drive a specific process during the calculation, such as the treatment of the boundary conditions. For this reason, every mesh builder in `P2MESH` takes input pointers to user defined functions capable of processing markers. Pointer names are indicated by

- `mark_vertex`
- `mark_edge`
- `mark_poly`

When the mesh is internally generated, a set of markers following an internal convention – described in a later remark – is automatically created. The markers indicate the location of the internally generated vertices, edges and polygons of the mesh to the external marker processing functions. Two examples will illustrate how markers work in `P2MESH`.

The simplest action a program can do with a marker – apart from ignoring it – is to save its value somewhere. In the source fragment which follows, markers are read from an external file by the mesh builder `read_mesh`, but if they were internally generated, the case would be the same.

```
//define class Vertex
class Vertex : public p2_vertex<Common> {
  unsigned v_marker ;
public:
  static void Set_BC(Vertex & v, unsigned const & marker)
  { v . v_marker = marker ; }
} ;

//define class Edge
class Edge : public p2_edge<Common> {
  unsigned e_marker ;
public:
  static void Set_BC(Edge & e, unsigned const & marker)
  { e . e_marker = marker ; }
} ;

//define class Triangle
class Triangle : public p2_poly<Common> {
  unsigned t_marker ;
public:
  static void Set_BC(Triangle & t, unsigned const & marker)
  { t . t_marker = marker ; }
} ;

class Mesh : public p2_mesh<Common> {
public:
  Mesh(char const file[])
  { read_mesh(file,
              Vertex::Set_BC,
              Edge::set_BC,
              Triangle::set_BC) ; }
}
```

Analysis The project classes Vertex, Edge, and Triangle are defined in order to contain a public integer attribute to be used for storing the value of the external marker. This action is specified in the definition of the corresponding marker processing function, which takes as input argument the reference to the current instance and the marker value. The function action consists in assigning the input marker value to the public marker attribute of the current geometric entity. The constructor `Mesh(char const file[])` invokes the mesh builder `read_mesh` which takes in input the pointers to the three marker processing function.

When the mesh is instantiated, the mesh builder `read_mesh` performs an internal loop over all the vertices, edges and triangles of the mesh data set, and for any entity the appropriate marker processing function is called. No action would occur if the entry `NULL` is given in input.

Markers may also be defined by the user as data with an internal structure. The following source fragment illustrate the situation:

```
# include "p2mesh.hh"

// declare the name of the user defined classes
class Vertex ;
class Edge ;
class Triangle ;
class Mesh ;

// define the marker type Marker
struct Marker {
  string name ;
  double value ;
} ;

// define the output operator <<
ostream &
operator << (ostream & s, Marker const & m)
{ s << "( " << m . name << " = " << m . value << " )" ; return s ; }

// define the input operator >>
istream &
operator >> (istream & s, Marker & m)
{ s >> m . name >> m . value ; return s ; }

// define the class Common with the user defined type Marker
class Common : public p2_common<Vertex,Edge,Quad,Mesh,
                                3,false,
                                double, int, unsigned,
                                Marker, Marker, Marker> {
  // .. user stuff
} ;
```

Analysis The marker type `Marker` is defined by a `struct` statement. It contains the string variable `name` and the double precision variable `value`. The user is also asked to implement the I/O operators >> and <<, whose definitions overload the built-in ones. The complex marker data type is then introduced in the P2MESH-based application by specifying it in the argument list of `p2_common` when the project class Common is publicly derived.

6.1.1 tensor_mesh

The method `tensor_mesh` generates a regular mesh within the rectangle specified by the coordinates of the bottom-left vertex `(xmin, ymin)` and the top-right vertex `(xmax, ymax)`:

```
void
tensor_mesh(
  double const & xmin, // bounding box of the mesh
  double const & xmax,
  double const & ymin,
  double const & ymax,
  unsigned const nx, // x-subdivision
  unsigned const ny, // y-subdivision
  void (*mark_vertex)(Vertex &, unsigned const &),// vertex marker
  void (*mark_edge)  (Edge   &, unsigned const &),// edge marker
  void (*mark_poly)  (Poly   &, unsigned const &),// polygon marker
  unsigned const kind = 0) // 0 or 1 based index vectors
```

6.1.2 std_tensor_mesh

The method `std_tensor_mesh` generates a regular mesh within the unit square box $[0,1] \times [0,1]$:

```
void
std_tensor_mesh(
  unsigned const nx, // x-subdivision
  unsigned const ny, // y-subdivision
  void (*mark_vertex)(Vertex &, unsigned const &),// vertex marker
  void (*mark_edge)  (Edge   &, unsigned const &),// edge marker
  void (*mark_poly)  (Poly   &, unsigned const &),// polygon marker
  unsigned const kind = 0) // 0 or 1 based index vectors
```

- The integer values `nx` and `ny` are the number of partitions in the `x` and `y` cartesian directions.

- The pointer functions `mark_vertex`, `mark_edge` and `mark_poly` allow to specify some actions on the geometrical entities at the mesh data set initialization phase, for example assigning boundary condition identifiers and so on. If one of this input entry is set to `NULL`, no action is performed on the corresponding set of entities. See also the following remarks on the internal marker convention and marker usage.

- The entry `kind` is specific to triangular meshes and is ignored on quadrilateral meshes. It allows to change the orientation of triangles in the mesh, see Figure 5.

6.1.3 map_mesh

The method `map_mesh` generates a regular mesh within a four-side domain which is the image of the unit square box by the user defined mapping function `shape`:

```
void
map_mesh(
  void (*shape) (double const & s, double const & t,
                 double & x, double & y) ,
  // shape function
  unsigned const ns, // s-subdivision
  unsigned const nt, // t-subdivision
  void (*mark_vertex)(Vertex &, unsigned const &),// vertex marker
  void (*mark_edge)  (Edge   &, unsigned const &),// edge marker
  void (*mark_poly)  (Poly   &, unsigned const &),// polygon marker
  unsigned const kind = 0) // 0 or 1 based index vectors
```

- The integer values `ns` and `nt` specify the number of partitions in the `s` and `t` cartesian directions.

6.1.4 read_map_mesh

The method `read_map_mesh` generates a regular mesh from the input triangulation in the ASCII file `file_name`:

```
void
read_map_mesh(
  char const * const file_name, // base name for file grid
  void (*mark_vertex)(Vertex &, unsigned const &),// vertex marker
  void (*mark_edge)  (Edge   &, unsigned const &),// edge marker
  void (*mark_poly)  (Poly   &, unsigned const &),// polygon marker
  unsigned const kind = 0) // 0 or 1 based index vectors
```

Example Figure 5 shows the grids that can be generated by different values of the parameter `kind` by the statement

```
tensor_mesh(0.0, 1.0, 0.0, 1.0, 4, 4, NULL, NULL, NULL, kind) ;
```

that is also equivalent to

```
std_tensor_mesh(4, 4, NULL, NULL, NULL, kind) ;
```

since in this case `xmin`, `xmax`, `ymin`, `ymax` are set to the standard square box $[0, 1] \times [0, 1]$.

Example The mapping function

```
void shape(double const & s, double const & t,
           double       & x, double       & y) {
  static double const PI2 = 2 * atan(1.0) ; // pi/2
  x = (1+t) * sin(s * PI2) ;
  y = (1+t) * cos(s * PI2) ;
}
```

maps the unit square onto the four-side domain shown in figure 6, that can then be triangulated by changing quadrilaterals or triangles – and different orientations for these latter ones – by choosing the appropriate value of `kind`.

Example A four-side domain of generic shape can also be triangulated starting from an input data file, containing an header line and the vertex coordinates of a basic grid. The header line requires two integer fields, `ns` and `nt`, that stands for the number of vertices in the s and t directions. The total number of vertices must be `ns`$\times$`nt`, and for

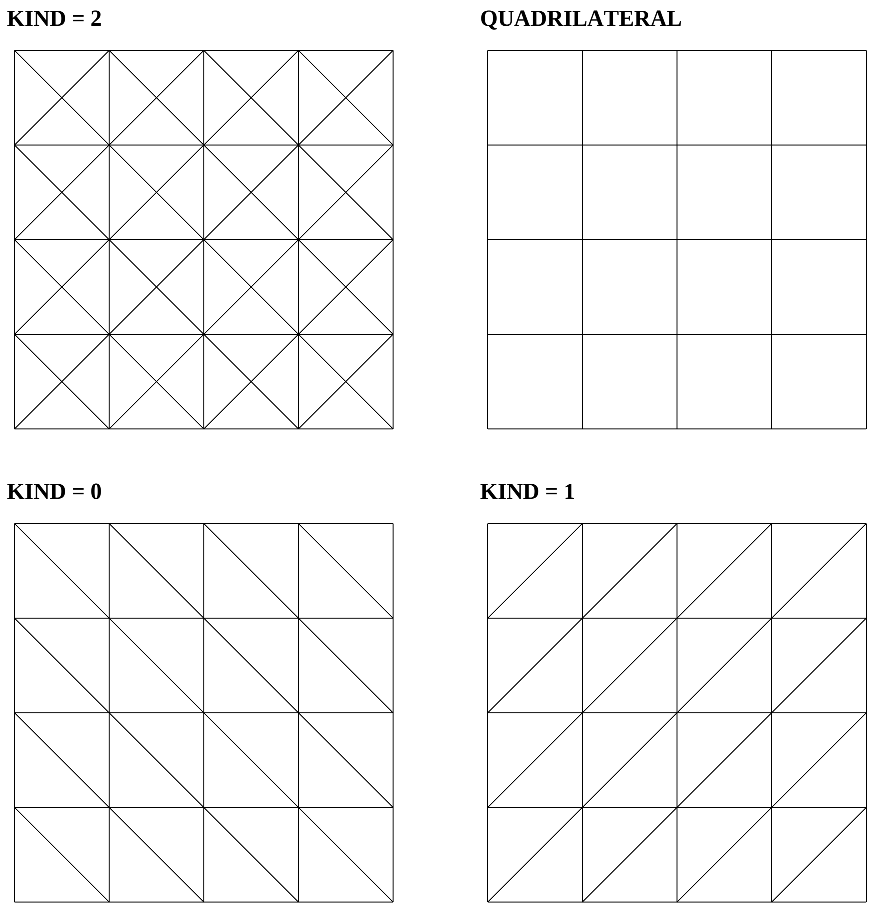

Figure 5: Grids built using `tensor_mesh`

any vertex an entry line in the file gives its first and second coordinates. The fastest running index is the one for the direction s.

For easy of use, comments may be introduced in the file by inserting any of the following symbols

“`!`”, “`#`”, “`;`”, “`%`”, “`$`”,

as the first character of the line; the rest of the line is then ignored.

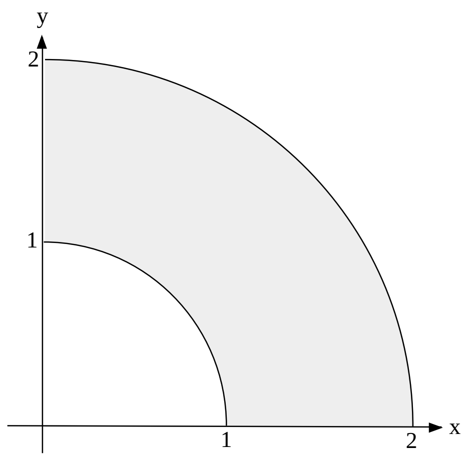

Figure 6:

Suppose the file `mesh.grd` contains, according to the previous specifications, the vertex coordinates of the grid produced by applying the mesh builder function `map_mesh` with the mapping function `shape` and `ns=4` and `nt=4`.

Then, the two statements

```
map_mesh( shape, 4, 4, NULL, NULL ,NULL, kind ) ;
```

and

```
read_map_mesh( "mesh.grd", NULL, NULL, NULL, kind ) ;
```

generates the grids in Figure 7:

Remark **(The internal convention for markers)**

When one of the previous mesh builders is executed, internal markers are automatically generated. The markers indicate the logical location onto the regular four-side regular grid of any instance of the project classes. Thus, they can be processed at the initialization phase of the mesh data set by invoking suitable user defined marker functions, whose action has to be coherently specified in the application program.

Internal markers are generated using the following convention.

file: mesh.grd

```
# ns+1 nt+1
3 3
# ns+1 nt+1
#  x      y
0.00   1.00
0.71   0.71
1.00   0.00
#
0.00   1.50
1.06   1.06
1.50   0.00
#
0.00   2.00
1.41   1.41
2.00   0.00
```

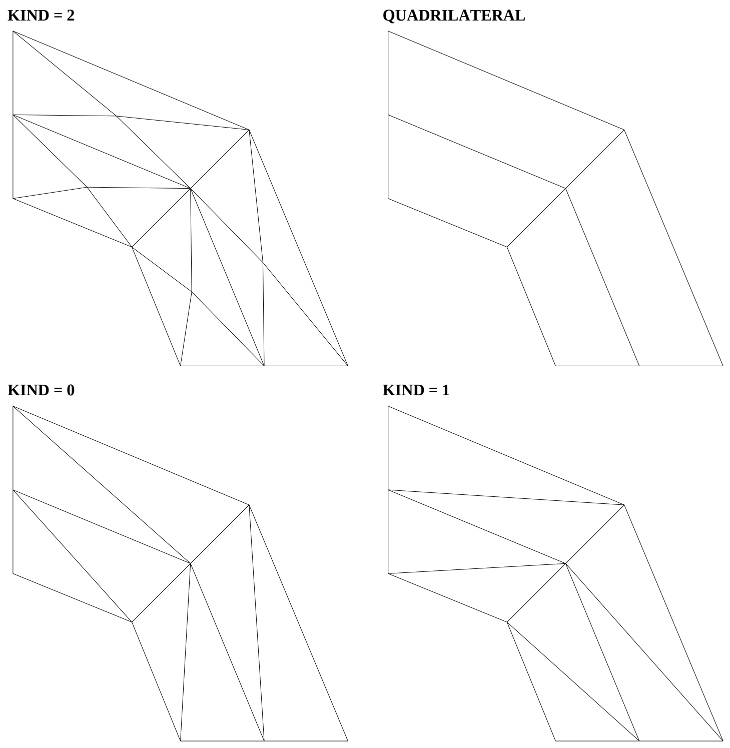

Figure 7: grid built using `map_mesh` and `red_map_mesh`

Vertex convention

0	*internal* vertex;
1	*bottom side* boundary vertex;
2	*right side* boundary vertex;
3	*top side* boundary vertex;
4	*left side* boundary vertex;
5	*bottom left* corner vertex;
6	*bottom right* corner vertex;
7	*top right* corner vertex;
8	*top left* corner vertex.

Edge convention

0	*internal* edge;
1	*bottom side* boundary edge;
2	*right side* boundary edge;
3	*top side* boundary edge;
4	*left side* boundary edge;

Element convention

0	*internal* element;
1	*bottom side* boundary element;
2	*right side* boundary element;
3	*top side* boundary element;
4	*left side* boundary element;
5	*bottom left* corner element;
6	*bottom right* corner element;
7	*top right* corner element;
8	*top left* corner element.

Figure 8 illustrates the internal convention in the case of a regular 3×3 triangle based mesh. It is worth noting that for all these mesh builders, the entity numbering proceeds

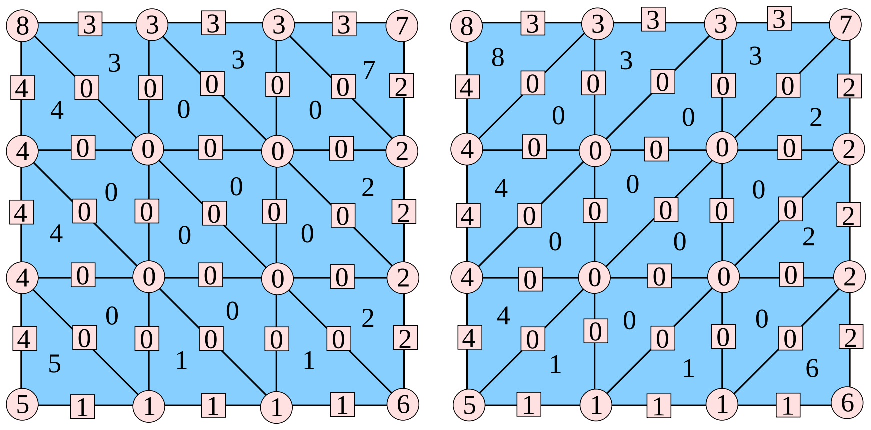

Figure 8: Marker internal convention

in a very rigid way. The numbering always starts from the most left-bottom located geometric entity; then, the boundary entities are first numbered following the external boundaries in a counterclockwise way and then the internal perimeters in a clockwise way; finally, all internal entities are numbered from left to right and bottom to up.

Hence, the location of every geometric entity on the mesh can also be determined by simply knowing its `local_number`, but this approach should demand for more expensive run-time *if-test* comparisons scattered in the code. Internal markers allow an easier manipulation of this information, at the cost of a very small redundancy, which can also be avoided by simply ignoring them whenever useless.

6.1.5 build_mesh

The method `build_mesh` generates a mesh from a topology description in memory. The prototype is:

```
void
build_mesh(
  unsigned const nv,
  double   const *XY,
  Vmark    const *mv,
  void mark_vertex(Vertex &, Vmark const &),

  unsigned const ne,
  unsigned const *E,
  Emark    const *me,
  void mark_edge(Edge &, Emark const &),

  unsigned const np,
  unsigned const *P,
  Pmark    const *mp,
  void mark_poly(Poly &, Pmark const &),

  unsigned const base = 0) ;
```

where

- `nv` total number of vertices;
- `XY` real array storing the vertex coordinates `x` and `y` in a sequential way, i.e.:

$$\mathsf{XY} = (x_0, y_0, x_1, y_1, \ldots, x_{nv}, y_{nv})$$

- `mv` integer array storing the vertex marker values; when markers are not to be specified, the entry `NULL` must be set;

- `mark_vertex` pointer to a user-defined marker routine; when markers are not to be specified, the entry `NULL` must be set;

- `ne` total number of edges;

- `E` integer array storing the edge connectivities, given sequentially by the pairs of pointers (e_i^a, e_i^b) to the position of the vertices e_i^a and e_i^b within the array `XY`, i.e.

$$\mathsf{E} = (e_0^a, e_0^b, e_1^a, e_1^b, \ldots, e_{ne}^a, e_{ne}^b);$$

- `me` integer array storing the edge marker values; when markers are not to be specified, the entry `NULL` must be set;

- `mark_edge` pointer to a user-defined marker routine; when markers are not to be specified, the entry `NULL` must be set;

- `np` total number of polygons;

- `P` integer array storing the vertex indices forming the polygon (`3` for a triangular mesh, `4` for a quadrilateral mesh); for example, in the former case `P` stores information as

$$\mathsf{P} = (t_0^a, t_0^b, t_0^c, t_1^a, t_1^b, t_1^c, \ldots, t_{np}^a, t_{np}^b, t_{np}^c)$$

  where (t_i^a, t_i^b, t_i^c) are the indices within the array `XY` of the vertex i.

- `mp` integer array storing the triangle marker values; when markers are not to be specified, the entry `NULL` must be set;

- `mark_poly` pointer to a user-defined marker routine; when markers are not to be specified, the entry `NULL` must be set;

- `base` is the offset of the connectivity array indexing. It must be explicitly set to `1` if the arrays are indexed from `1` in accord with the FORTRAN convention. Otherwise, the offset is `0` in accord with the C convention.

6.1.6 read_mesh

The public method `read_mesh` generates a mesh data set from an input ASCII file. The prototype of the method is

```
void
read_mesh(
  char const file_name[],
  void (*mark_vertex) ( Vertex &, Vmark const &),
  void (*mark_edge)   ( Edge   &, Emark const &),
  void (*mark_poly)   ( Poly   &, Pmark const &),
  unsigned const base = 0)
```

The method `read_mesh` can read three ASCII files with name `file_name` and extensions `*.node`, `*.ele`, and `*.edge`, in the output format of the freeware mesh generator `TRIANGLE`[2] . This format is briefly described in the following. Unless otherwise indicated in the format description the entry fields are mandatory.

file *.node `Mandatory`.
The file lists the vertex coordinates and, optionally, a vertex marker. The first line is a header with the number of vertices, which indicates also the total number of the remaining lines in the file. Each other line contains the following entries

1. the vertex identifier (one integer field);
2. the vertex coordinates (two floating point fields);
3. the vertex marker (one `Vmark` field, optional).

file *.ele `Mandatory`.
The file contains the node–element connectivity, and optionally, an element marker. The first line is a header with the number of elements, which indicates also the total number of the remaining lines in the file. Each other line contains the following entries

1. the element identifier (one integer field);
2. the identifiers of the vertices in the current element (as many integer fields as vertices in the polygon);
3. the element marker (one `Pmark` field, optional).

[2]http://www.cs.cmu.edu/afs/cs.cmu.edu/project/quake/public/www/triangle.html

file *.edge `Optional`.
If the file is present, it contains the edge–vertex connectivity, and optionally an edge marker. If absent, the edge–vertex connections are internally detected and no edge marker is assigned. The first line is a header with the number of edges, which indicates also the total number of the remaining lines in the file. Each other line contains the following entries

1. the edge identifier (one integer field);
2. the identifiers of the vertices connected by the current edge (two integer fields);
3. the edge marker (one `Emark` field, optional).

If the first character of the lines in the files is one of the following:

“`!`”, “`#`”, “`;`”, “`%`”, “`$`”,

the rest of the line is ignored as a comment. Notice that this format is slightly more general than the one used by triangle, because instead of an integer number the marker can be an object of a different type defined in the project.

Example The following two files `box.node` and `box.ele` are given by

file: box.node

```
# vertices (no marker)
9
# nv      x     y
   1    0.0   0.0
   2    0.5   0.0
   3    0.5   0.5
   4    1.0   0.5
   5    1.0   1.0
   6    0.0   1.0
   7    0.25  0.5
   8    0.5   0.75
   9    0.5   1.0
```

file: box.ele

```
# triangles (no marker)
8
# nt  v0  v1  v2
   1   1   2   7
   2   6   7   8
   3   8   5   9
   4   2   3   7
   5   3   4   8
   6   8   9   6
   7   5   8   4
   8   3   8   7
```

Table 1: Example of file mesh definition

The use of the file “`box`” in input of `read_mesh`

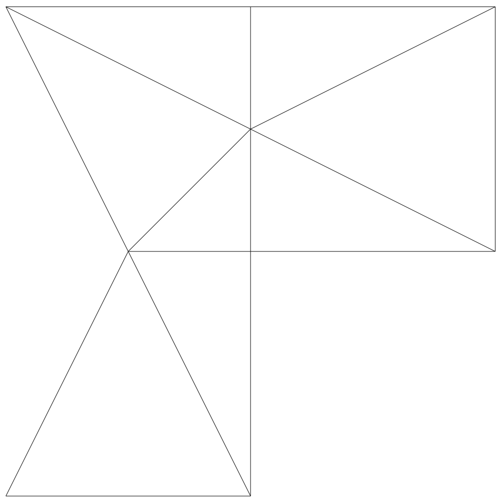

Figure 9: Generated mesh

```
read_mesh("box", NULL, NULL, NULL, 1) ;
```

produces the mesh in figure 9.

6.2 Statistics and diagnostics

The two following methods respectively print some simple information about the mesh data set generated by one of the previous mesh builders and check the consistency of the data set.

n.	Method	Description
84	`void report(ostream& s)`	generate mesh statistics
85	`bool test_mesh()`	check the mesh consistency.

The output of method (84) is sent to the output stream `s`. For example, the statement `report(cout)` produces on the standard output stream the following info about the mesh defined in Table 1 on page 47:

```
             p2_mesh statistics
             Polygon Type = Triangle
            +----------+----------+----------+
            | Total    | Internal | Boundary |
+----------+----------+----------+----------+
| Vertices |       14 |        5 |        9 |
+----------+----------+----------+----------+
| Edges    |       30 |       21 |        9 |
+----------+----------+----------+----------+
| Polygons |       17 |        8 |        9 |
+----------+----------+----------+----------+
```

When a mesh is generated from an external file, the consistency of the mesh should be ensured by the final user of the application program. However, when the files containing connectivities are corrupted, results are unpredictable and errors may be difficult to detect.

A number of errors are signaled during the reordering procedure, but many other may be hidden. Method (85) checks the internal consistency of the mesh data set generated during the initialization phase. When invoked, the method `test_mesh()` performs

- a test on the connectivities of polygons, edges, and vertices in the data set;
- a test on the orientation of the vertices in any polygon list, which must be ran in a counterclockwise manner; an error in orientation is responsible of negative polygon areas;
- a test on the connectivity of the vertex lists, when active;
- a test to detect eventual not referenced instances in the data set.

This method, although not too much expensive in terms of CPU time should not be used in normal situations.

The following methods prints information contained in the mesh objects.

n.	Method	Description
86	`void print(ostream & s, Unsigned base)`	print the mesh in a human readable form. The numbering starts from the offset `base`.

87	`void print(ostream & s,`
	`           Vertex & V,`
	`           unsigned base)`
		print the contents of vertex `V` in a human readable form.

88	`void print(ostream & s,`
	`           Edge & E,`
	`           unsigned base)`
		print the contents of edge `E` in a human readable form.

89	`void print(ostream & s,`
	`           Poly & P,`
	`           unsigned base)`
		print the contents of polygon `P` in a human readable form.

7 STL Iterators

Description An iterator is an object that iterates over the mesh lists of vertices, edges, and polygons in a very effective and transparent way. Its use is highly recommended, since it hides all the details of how the sequence of items on which we wish to iterate is actually stored and accessed. Hence, the user application is implemented to be truly independent of both the platform and of the language version. Compatibility is also ensured with any future library developments.

The `P2MESH` software library has been designed using as much as possible the `STL` technology. The mesh iterators in `P2MESH` are thus inherited from the ones given in `STL`.

7.1 Vertex iterators

There are *two* iterator types derived from `STL`:

- `vertex_iterator` iterates on `Vertex`-type objects;
- `vertex_const_iterator` iterates on constant `Vertex`-type objects;

The iterator types are defined as public ones but in the scope of the class Mesh. To access them, the scope operator must be used, e.g.

`Mesh::vertex_iterator`.

The following methods may be used to build loops on the vertices of the current mesh data set. They return an iterator object, which, as indicated, points either to the first or to the past-to-last instance in the internal mesh representation of the vertex data set.

n.	Method	Description

90	`vertex_iterator vertex_begin()`
	`vertex_const_iterator vertex_begin()`
		smart pointer to the first vertex

91	`vertex_iterator vertex_end()`
	`vertex_const_iterator vertex_end()`
		smart pointer to the past-to-last vertex

92	`vertex_iterator ivertex_begin()`
	`vertex_const_iterator ivertex_begin()`
		smart pointer to the first internal vertex

93	`vertex_iterator ivertex_end()`
	`vertex_const_iterator ivertex_end()`
		smart pointer to the past-to-last internal vertex

94	`vertex_iterator bvertex_begin()`
	`vertex_const_iterator bvertex_begin()`
		smart pointer to the first boundary vertex

95	`vertex_iterator bvertex_end()`
	`vertex_const_iterator bvertex_end()`
		smart pointer to the past-to-last boundary vertex

Usage The following example illustrates how these iterators may be used.

```
Mesh my_mesh ; // create a mesh object
// do something...
Mesh::vertex_iterator iv ;       // define an iterator on vertex data
for ( iv = mesh.ivertex_begin() ;// points to the first internal edge
      iv != mesh.ivertex_end() ; // is it the last vertex ?
      ++iv) {                    // advance to the next vertex

     Vertex & V = *iv ;   // the current vertex reference
     Vertex * pV = &*iv ; // pointer to the current vertex
     /*
     ... do something on the vertex referenced by V or pointed by pV
     */
```

7.2 Edge iterators

There are *two* iterator types derived from STL:

- `edge_iterator` iterates on `Edge`-type objects;
- `edge_const_iterator` iterates on constant `Edge`-type objects;

The iterator types are defined as public ones but in the scope of the class Mesh. In order to access them, the scope operator must be used, e.g. `Mesh::edge_iterator`.

The following methods may be used to build loops on the edges of the current mesh data set. They return an iterator object, which, as indicated, points either to the first or to the past-to-last instance in the internal mesh representation of the edge data set.

n.	Method	Description

96	`edge_iterator edge_begin()`
	`edge_const_iterator edge_begin()`
		smart pointer to the first edge

97	`edge_iterator edge_end()`
	`edge_const_iterator edge_end()`
		smart pointer to the past-to-last edge

98	`edge_iterator iedge_begin()`
	`edge_const_iterator iedge_begin()`
		smart pointer to the first internal edge

99	`edge_iterator iedge_end()`
	`edge_const_iterator iedge_end()`
		smart pointer to the past-to-last internal edge

100	`edge_iterator bedge_begin()`
	`edge_const_iterator bedge_begin()`
		smart pointer to the first boundary edge

101	`edge_iterator bedge_end()`
	`edge_const_iterator bedge_end()`
		smart pointer to the past-to-last boundary edge

Usage The following example illustrates how these iterators may be used.

```
Mesh my_mesh ; // create a mesh object
// do something...
Mesh::edge_iterator ie ;       // define an iterator on edge data
for ( ie = mesh.edge_begin() ; // points to the first edge
      ie != mesh.edge_end() ;  // is it the last edge ?
      ++ie) {                  // advance to the next edge

     Edge & E = *ie ;   // the current edge reference
     Edge * pE = &*ie ; // pointer to the current edge
     /*
     ... do something on the edge referenced by E or pointed by pE
     */
```

7.3 Poly Iterators

There are *two* iterator types derived from STL:

- `poly_iterator` iterates on `Poly`-type objects;
- `poly_const_iterator` iterates on constant `Poly`-type objects;

The iterator types are defined as public ones but in the scope of the class Mesh. In order to access them, the scope operator must be used, e.g. `Mesh::poly_iterator`.

n.	Method	Description

102	`poly_iterator poly_begin()`
	`poly_const_iterator poly_begin()`
		smart pointer to the first polygon

103	`poly_iterator poly_end()`
	`poly_const_iterator poly_end()`
		smart pointer to the past-to-last polygon

104	`poly_iterator ipoly_begin()`
	`poly_const_iterator ipoly_begin()`
		smart pointer to the first internal polygon

105	`poly_iterator ipoly_end()`
	`poly_const_iterator ipoly_end()`
		smart pointer to the past-to-last internal polygon

106	`poly_iterator bpoly_begin()`
	`poly_const_iterator bpoly_begin()`
		smart pointer to the first boundary polygon

107	`poly_iterator bpoly_end()`
	`poly_const_iterator bpoly_end()`
		smart pointer to the past-to-last internal polygon

Usage The usage of this iterators is quite standard

```
Mesh my_mesh ; // create a mesh object
// do something...
Mesh::poly_iterator ip ;       // define an iterator on polygons
for ( ip = mesh.bpoly_begin() ;// points to the first boundary item
      ip != mesh.bpoly_end() ; // is it the last polygon ?
      ++ip) {                  // advance to the next polygon

     Poly & P = *ip ;   // the current poly reference
     Poly * pP = &*ip ; // pointer to the current poly
     /*
     ... do something on the poly referenced by P or pointed by pP
     */
}
```

8 Iterators public interface

Class Name Iterator<T>,
CIterator<T>.
Throughout the section the keyword T will generically indicates one of the three user defined classes Vertex, Edge, Poly, (not necessarily the same at any occurrence).

Description The iterators previously introduced and directly derived from the STL are sufficient to implement any kind of loops in numerical algorithms. However, a different set of iterators is available in P2MESH that allows, without any loss of efficiency, the implementation of more readable source codes. Iterator objects are instantiated by a template definition, which requires the type of the data set to be iterated on. The current mesh to whom the data set belongs and the iteration range may be specified either as arguments of the iterator constructor methods or set in a subsequent moment by invoking the method set_loop. The classes Iterator<T> and CIterator<T> have the same functionality, the only difference is that the class CIterator<T> iterates over **constant** objects.

Usage The following piece of code illustrates how iterators can be instantiated in the code.

```
Iterator<Vertex> v_iter; // instantiate an iterator on mesh vertices
Iterator<Edge>   e_iter; // instantiate an iterator on mesh edges
Iterator<Poly>   p_iter; // instantiate an iterator on mesh polygons
```

Member Functions Several iterator constructors are supported by P2MESH. The following table reports the different cases. M indicates the current mesh data set, and f=1,2,3 is a flag which specifies the iteration range.

n.	Constructor	Description
108	`Iterator<T> (Mesh & M)`	iterator for the mesh M
109	`Iterator<T> (Mesh & M, unsigned f)`	iterator for the mesh M on the range f
110	`Iterator<T> (void)`	iterator, no mesh and range specified

The mesh data set and the iteration range can be set up after the iterator was instantiated by the public methods in the table.

n.	Method	Description
111	`void set_loop(Mesh & M)`	select the mesh data set `M`
112	`void set_loop(Mesh & M, unsigned f)`	select the mesh data set `M` and the range `f`
113	`void begin()`	set iterator to the first item
114	`bool end_of_loop()`	return true if all items were iterated
115	`Iterator<T> const & operator ++ ()`	advance the iterator to the next item

116	`T const * operator () () const`
	`T       * operator () ()`
		return the current item of the iterator

The methods (108–109) accept in input as a first entry the reference to the given instance of the class `Mesh`, indicated by `M`. The method `set_loop` in the forms (112) has an optional flag indicating the iteration range

- `f=0` : iterates over all the items in the mesh data set;
- `f=1` : iterates on the boundary items in the mesh data set;
- `f=2` : iterates on the internal items in the mesh data set;

Constructor (108) assumes that `f=0` as the default state for the iteration range. Constructor (110), instead, does not assume any default state; hence, the iteration range is undefined until directly specified by the user application, for example by a later usage of the method `set_loop()`. An iteration loop can be finally built by using the public methods (113–116). The following piece of code illustrates an example of how iterators can be instantiated and used in an application program.

```
Mesh my_mesh ; // create a mesh object
// do something...
Iterator<Edge> edge_iterator ;        // an iterator on edge data
edge_iterator . set_loop(my_mesh,2) ; // loops on internal edges
for ( edge_iterator.begin() ;         // points to the internal edge
      ! edge_iterator.end_of_loop() ; // it is not the last edge
      ++edge_iterator) {              // advance to the next edge

     Edge & E = *edge_iterator ; // the current edge reference
     Edge * pE = &*edge_iterator ; // pointer to the current edge
     /*
     ... do something on the edge referenced by E
     */
}
```

The macro ***`foreach`.*** A special macro, named `foreach`, is available. It is defined by the preprocessor statement

```
# define foreach(X) for ( X . begin() ; ! X . end_of_loop() ; ++X )
```

and makes possible a very short and effective definition of loops, as shown by the following example.

```
// user stuff...

# include "p2mesh.hh"

// user stuff...

void main() {
  Mesh my_mesh ; // create a mesh object
  /*
   do something...
   */
  // define an iterator and set it to loop on internal edges
  Iterator<Edge> internal_edge(my_mesh, 2) ;

  foreach( internal_edge ) {   // loops on internal edges
     Edge & E = *internal_edge ; // the current edge reference
     Edge * pE = &*internal_edge ; // pointer to the current edge
     /*
     ... do something on the edge referenced by E
     */
  }
  // other stuff ...
}
```

In order to avoid conflicts with other libraries, which may define macros with the same name for similar purposes, it is possible to turn off the macro `foreach` by the following preprocessor directive

```
# define P2MESH_NO_FOREACH
```

Three preprocessor `define` statements can be used in `P2MESH` *before* the inclusion of the header file of the library. They are:

- `P2MESH_DEBUG`
- `P2MESH_VERBOSE`
- `P2MESH_NO_FOREACH`

The following piece of code

```
# define P2MESH_DEBUG
# include "p2mesh.hh"
```

forces the library code to check if subscripts are within their bounds during execution of internal loops.

The verbose mode is switched by

```
# define P2MESH_VERBOSE
# include "p2mesh.hh"
```

and produces run-time info messages on standard output about `P2MESH` internal operations.

Finally, the macro `foreach` can be switched off by the following source fragment

```
# define P2MESH_NO_FOREACH
# include "p2mesh.hh"
```

A – Files included by P2MESH

P2MESH includes the STL header files:

- algo
- vector

the C++ header files:

- fstream
- iomanip
- iostream
- string

and the standard C header file:

- stdlib.h

B – Error indicators and warnings

This section documents the run-time diagnostics messages which are output by `P2MESH`. The symbol "•" indicates that an internal control is always performed by the library code. Otherwise, the symbol "○" indicates that the control is detected and printed only when the preprocessor directive `# define P2MESH_DEBUG` is given explicitly.

B.1 Diagnostics Messages of the class `p2_vertex`

•
```
P2MESH in method ‘‘p2_vertex::operator =’’
Fatal error: attempt to use copy constructor.
```

This error message is given on return when the user application program attempts to invoke the default copy constructor provided by the C++ programming language. The failure is motivated in order to force the user to perform a (safer) bitwise copy.

○
```
P2MESH in method ‘‘p2_vertex::vertex( #1 )’’
Fatal error: local index out of range.
```

An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_vertex::n_vertex()-1]`.

•
```
P2MESH in method ‘‘p2_vertex::vertex( #1 )’’
Fatal error: vertex list not defined.
```

This error message is output whenever the user tries to access the list of adjacent vertices but the argument `List` in the template header definition of the project class Common is set to `false` (default).

○
```
P2MESH in method ‘‘p2_vertex::edge( #1 )’’
Fatal error: local index out of range.
```

An invalid index was detected on entry. The correct value must be an integer in the range `[0...p2_vertex::n_edge()-1]`.

•
```
P2MESH in method ‘‘p2_vertex::edge( #1 )’’
Fatal error: edge list not defined.
```

This error message is output whenever the user tries to access the list of adjacent edges but the argument `List` in the template header definition of the project class Common is set to `false` (default).

- ○ `P2MESH in method ‘‘p2_vertex::poly( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer in the range `[0...p2_vertex::n_poly()-1]`.

- ● `P2MESH in method ‘‘p2_vertex::poly( #1 )’’`
  `Fatal error: polygon list not defined.`

  This error message is output whenever the user tries to access the list of adjacent polygons but the argument `List` in the template header definition of the project class Common is set to `false` (default).

B.2 Diagnostics Messages of the class p2_edge

- ● `P2MESH in method ‘‘p2_edge::operator =’’`
  `Fatal error: attempt to use copy constructor.`

  This error message is produced when the user application program attempts to invoke the default copy constructor provided by the C++ programming language. The failure is motivated to force the user to perform a (safer) bitwise copy.

- ○ `P2MESH in method ‘‘p2_edge::vertex( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_edge::n_vertex()-1]`.

- ○ `P2MESH in method ‘‘p2_edge::edge( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_edge::n_edge()-1]`.

- ○ `P2MESH in method ‘‘p2_edge::poly( #1 )’’`
  `Fatal error: local index out of range.`

An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_edge::n_poly()-1]`.

- ○ `P2MESH in method ‘‘p2_edge::x( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be `0` or `1`.

- ○ `P2MESH in method ‘‘p2_edge::y( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be `0` or `1`.

B.3 Diagnostics Messages of the class `p2_poly`

- • `P2MESH in method ‘‘p2_poly::operator =’’`
  `Fatal error: attempt to use copy constructor.`

  This error message is given on return when the user application program attempts to invoke the default copy constructor provided by the C++ programming language. The failure is motivated in order to force the user to perform a (safer) bitwise copy.

- ○ `P2MESH in method ‘‘p2_poly::ok_oriented( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be `0` or `1`. An invalid index was detected on entry. The correct value must be `0` or `1`.

- ○ `P2MESH in method ‘‘p2_poly::vertex( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_poly::n_vertex()-1]`.

- ○ `P2MESH in method ‘‘p2_poly::edge( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_poly::n_edge()-1]`.

- ○ `P2MESH in method ‘‘p2_poly::poly( #1 )’’`

`Fatal error: local index out of range.`

An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_poly::n_poly()-1]`.

- ○ `P2MESH in method ‘‘p2_poly::poly( #1 )’’`
  `Fatal error: invalid polygon access.`

  The user application attempts to access to an adjacent polygon which does not exist. The polygon `poly(i)` does not exist when `edge(i)` is a boundary edge.

- ○ `P2MESH in method ‘‘p2_poly::ok_poly( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_poly::n_poly()-1]`.

- ● `P2MESH in method ‘‘p2_poly::local_number(vertex &)’’`
  `Fatal error: bad reference.`

  The vertex reference given to the method `local_number` as input argument does not correspond to a vertex objects in the list of vertices defining the current polygon instance.

- ● `P2MESH in method ‘‘p2_poly::local_number(edge &)’’`
  `Fatal error: bad reference`

  The edge reference given to the method `local_number` as input argument does not correspond to an edge objects in the list of edges defining the current polygon instance.

- ● `P2MESH in method ‘‘p2_poly::local_number(poly &)’’`
  `Fatal error: bad reference.`

  The poly reference given to the method `local_number` as input argument does not correspond to a poly objects in the list of polygons adjacent to the current polygon instance.

- ○ `P2MESH in method ‘‘p2_poly::x( #1 )’’`
  `Fatal error: local index out of range.`

  An invalid index was detected on entry. The correct value must be `0` or `1`.

- ○ `P2MESH in method ‘‘p2_poly::y( #1 )’’`
  `Fatal error: local index out of range.`

An invalid index was detected on entry. The correct value must be `0` or `1`.

-
  ```
  P2MESH in method ‘‘p2_poly::xm( #1 )’’
  Fatal error: local index out of range.
  ```
  An invalid index was detected on entry. The correct value must be `0` or `1`.

-
  ```
  P2MESH in method ‘‘p2_poly::ym( #1 )’’
  Fatal error: local index out of range.
  ```
  An invalid index was detected on entry. The correct value must be `0` or `1`.

-
  ```
  P2MESH in method ‘‘p2_poly::length( #1 )’’
  Fatal error: local index out of range.
  ```

B.4 Diagnostics messages of the class p2_mesh

-
  ```
  P2MESH in method ‘‘p2_mesh::vertex( #1 )’’
  Fatal error: global index out of range.
  ```
  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_mesh::n_vertex()-1]`.

-
  ```
  P2MESH in method ‘‘p2_mesh::edge( #1 )’’
  Fatal error: global index out of range.
  ```
  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_mesh::n_edge()-1]`.

-
  ```
  P2MESH in method ‘‘p2_mesh::poly( #1 )’’
  Fatal error: global index out of range.
  ```
  An invalid index was detected on entry. The correct value must be an integer within the range `[0...p2_mesh::n_poly()-1]`.

B.5 Build mesh error indicators

The internal methods `BuildEdges()`, `JointEdges()` and `Reorder()` are used by `P2MESH` to construct edges from a list of polygon connectivities and to reorder them by the algorithm depicted in the kernel description [1]. When a problem is detected in run-time initialization, a diagnostics message is produced. If such a situation occurs, the input mesh is more likely corrupted.

- `P2MESH in method ‘‘p2_mesh::BuildEdges()’’`
  `Fatal error:`
  `an edge is referenced twice from the same side.`

  An edge is shared by two different polygons which are located at the same side. In a correct mesh definition, an edge may be shared by no more than two polygons, which must be located at the two opposite sides of the edge.

- `P2MESH in method ‘‘p2_mesh::JointEdges()’’`
  `Fatal error: incomplete polygon found.`

  In the list of vertices of the current polygon one or more vertex references are `NULL`, i.e. they are not assigned to a valid vertex objects.

- `P2MESH in method ‘‘p2_mesh::JointEdges()’’`
  `Fatal error: try to build a polygon with a not existing edge.`

  The mesh data set misses the edge specified by two consecutive vertices in the vertex list of the current polygon.

- `P2MESH in method ‘‘p2_mesh::JointEdges()’’`
  `Fatal error: try to assign a polygon to an already assigned edge side.`

  `JointEdges()` attempts to assign more than one polygon to the same side of an edge.

- `P2MESH in method ‘‘p2_mesh::Reorder()’’`
  `Fatal error: isolated edge found.`

  An edge is found that does not belong to any polygon in the mesh data set.

- `P2MESH in method ‘‘p2_mesh::Reorder()’’`
  `Fatal error: incomplete edge found.`

  An edge is found with one or both `NULL` vertices.

- `P2MESH in method ‘‘p2_mesh::Reorder()’’`
  `Fatal error: corrupted boundary.`

The mesh is corrupted, because a boundary edge seems to be shared by two different boundaries.

-
  ```
  P2MESH in method ‘‘p2_mesh::Reorder()’’
  Fatal error: open boudary.
  ```

  The mesh is corrupted, because a chain of boundary edges cannot be closed.

B.6 Diagnostics messages for `read_map_mesh`

-
  ```
  P2MESH in method ‘‘p2_mesh::read_map_mesh(...)’’
  Fatal error: cannot open input file.
  ```

  An error occurred in opening the data file.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_map_mesh(...)’’
  Fatal error: bad grid dimension.
  ```

  An error occurred in reading the data file of a tensor map mesh; notice that the grid must be $n \times m$ with $n, m > 1$.

B.7 Diagnostics messages for `build_mesh`

-
  ```
  P2MESH in method ‘‘p2_mesh::build_mesh(...)’’
  Fatal error: bad edge definition in edge list.
  ```

  The method attempts to build an edge composed by two vertices, one or both of whose do not exist in the vertex list.

-
  ```
  P2MESH in method ‘‘p2_mesh::build_mesh(...)’’
  Fatal error: bad polygon definition in polygon list.
  ```

  The method attempts to assign an invalid vertex to a polygon.

B.8 Diagnostics Messages for `read_mesh`

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: cannot open nodes file.
  ```
  An error occurred in opening the node file.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: cannot open polygons file.
  ```
  An error occurred in opening the polygon file.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: error in reading vertex coordinates.
  ```
  An invalid vertex coordinate is found or premature end-of-file is reached.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: error in reading polygon definitions.
  ```
  An invalid polygon number is found or premature end-of-file is reached.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: error in reading vertex numbers for the polygon.
  ```
  Invalid vertex numbers for the definition of a polygon are found or premature end-of-file is reached.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: error in reading edge definition.
  ```
  An invalid edge number is found or premature end-of-file is reached.

-
  ```
  P2MESH in method ‘‘p2_mesh::read_mesh(...)’’
  Fatal error: error in reading vertex numbers for the edge.
  ```
  Invalid vertex numbers for the definition of an edge are found.

References

[1] Bertolazzi, E., and Manzini, G. The kernel of P2MESH. Tech. Rep. IAN–1166, IAN – CNR, 1999.