ISTITUTO
DI
ANALISI NUMERICA

del

CONSIGLIO NAZIONALE DELLE RICERCHE
via Abbiategrasso 209 – 27100 PAVIA (Italy)

PAVIA
1999

PUBBLICAZIONI

N. 1167

Enrico Bertolazzi, Gianmarco Manzini

P2MESH: Short Reference Guide

P2MESH: Short Reference Guide

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

Contents

(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.

The public interfaces

The following table indicates the names of the P2MESH library classes, the conventional names adopted in the chapter 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 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, in the examples we use their default values double, int and unsigned. 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 default type is unsigned.

p2_common public interface

Internal Prototype

```
template <typename P2V_type,
          typename P2E_type,
          typename P2P_type,
          typename P2M_type,
          unsigned SIZE_value    = 3,
          bool     LIST_value    = false,
          typename REAL_type     = double,
          typename INTEGER_type  = int,
          typename UNSIGNED_type = unsigned,
          typename VMARK_type    = unsigned,
          typename EMARK_type    = unsigned,
          typename PMARK_type    = unsigned>
class p2_common ;
```

Usage

A triangular mesh is specified by the following code fragment:

```
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
} ;
```

Hence, if vertex connectivity are demanded for a triangular mesh, the following code fragment must be used.

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

The following code fragment shows how different numerical types can parameterize all the internal implementation of the library. Floating-point real numbers are defined by the high-precision type `doubledouble`[1], and `long` and `unsigned long` instead respectively of `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 default type is `unsigned`. For example the following code fragment shows how `double` markers can be used instead of default `unsigned` markers.

```
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.

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

p2_vertex public interface

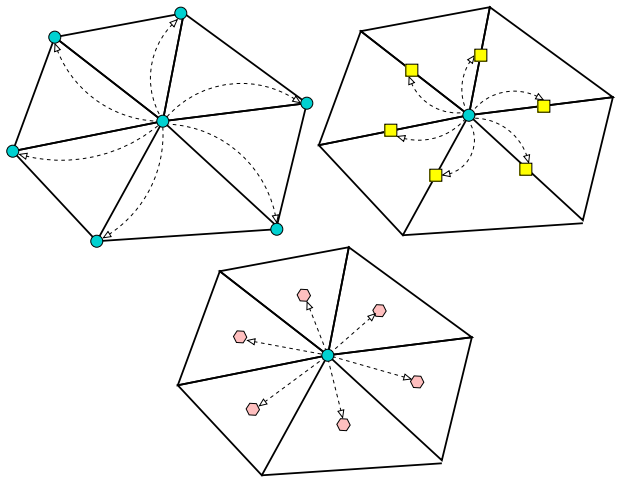

Figure 1: Vertex–Vertex, Vertex–Edge and Vertex-Polygon connections in the definition of a p2_vertex instance (pointers are optionally stored in memory)

Usage

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

Topological Methods

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`

Geometrical Methods

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

Topological 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`.

p2_edge public interface

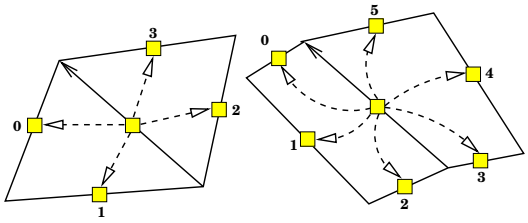

Figure 2: Surrounding edge numbering for triangles and quadrilaterals

Usage

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

Topological Methods

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

Geometrical Methods

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

p2_poly public interface

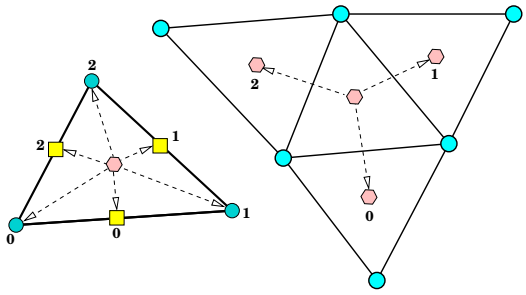

Figure 3: Polygon-Edge and Polygon-Polygon connections for a triangular p2_poly instance

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

Topological Methods

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

Geometrical Methods

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

Mapping Methods

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

p2_mesh public interface

Usage

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

Topological 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

Method n. 78 This method 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
```

Method n. 79 This method 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
```

Method n. 80 This method 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
```

Method n. 81 This method 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
```

- The integer values `nx` and `ny` specify 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.

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

The internal convention for markers

When one of the previous mesh builders is executed, internal markers are automatically generated. The markers indicate the logical position 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 is to be coherently specified in the application program.

Internal markers are generated using the following convention.

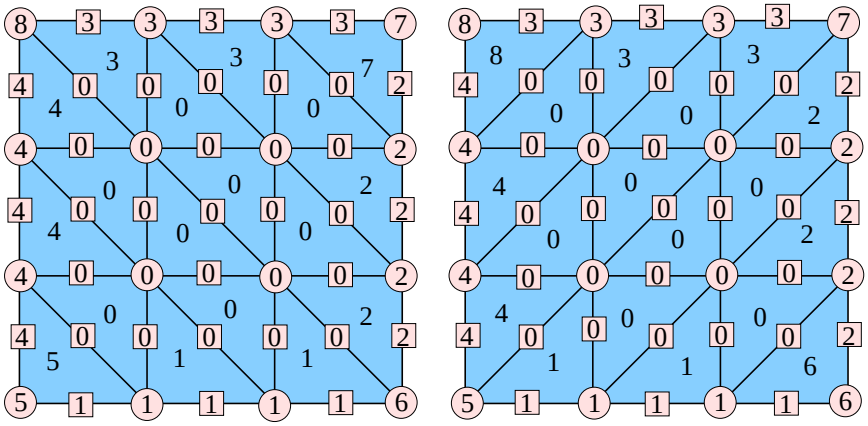

Figure 4: Marker internal convention

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.

Method n. 82 This method generates a mesh from a topology description in memory:

```
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) ;
```

- 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.

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

- `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

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

  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.

Method n. 83 This method 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 can read three ASCII files with name `file_name` and extensions `*.node`, `*.ele`, and `*.edge`. The method reads the output format of the freeware available mesh generator `TRIANGLE`[2] which is briefly described in the following. If not otherwise indicated, the entry fields in the file lines specified in the format description 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 (one integer field), 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 table, and optionally, an element marker. The first line is a header with the number of elements (one integer field), 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`.
The file, if present, contains the edge–vertex connectivity table, and optionally an edge marker. If absent, the edge–vertex connections are internally built and no edge marker is assigned. There is no other way to specify an edge marker to an external input mesh. The first line is a header with the number of edge (one integer field), 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 skipped as a comment.

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.

Printing

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.

STL Iterators

Description

An iterator is a particular object which allows to iterate the mesh data set list 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.

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. In order to access them, the scope operator must be used, e.g. `Mesh::vertex_iterator`.

n.	Method		Description
90	`vertex_iterator`	`vertex_begin()`	smart pointer to the first vertex
	`vertex_const_iterator`	`vertex_begin()`	
91	`vertex_iterator`	`vertex_end()`	smart pointer to the past-to-last vertex
	`vertex_const_iterator`	`vertex_end()`	
92	`vertex_iterator`	`ivertex_begin()`	smart pointer to the first internal vertex
	`vertex_const_iterator`	`ivertex_begin()`	
93	`vertex_iterator`	`ivertex_end()`	smart pointer to the past-to-last internal vertex
	`vertex_const_iterator`	`ivertex_end()`	
94	`vertex_iterator`	`bvertex_begin()`	smart pointer to the first boundary vertex
	`vertex_const_iterator`	`bvertex_begin()`	
95	`vertex_iterator`	`bvertex_end()`	smart pointer to the past-to-last boundary vertex
	`vertex_const_iterator`	`bvertex_end()`	

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`.

n.	Method		Description
96	`edge_iterator`	`edge_begin()`	smart pointer to the first edge
	`edge_const_iterator`	`edge_begin()`	
97	`edge_iterator`	`edge_end()`	smart pointer to the past-to-last edge
	`edge_const_iterator`	`edge_end()`	
98	`edge_iterator`	`iedge_begin()`	smart pointer to the first internal edge
	`edge_const_iterator`	`iedge_begin()`	
99	`edge_iterator`	`iedge_end()`	smart pointer to the past-to-last internal edge
	`edge_const_iterator`	`iedge_end()`	
100	`edge_iterator`	`bedge_begin()`	smart pointer to the first boundary edge
	`edge_const_iterator`	`bedge_begin()`	
101	`edge_iterator`	`bedge_end()`	smart pointer to the past-to-last boundary edge
	`edge_const_iterator`	`bedge_end()`	

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

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).

Usage

The following source fragment 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 method `set_loop` has an optional flag, which indicates iteration range

- `f=0` : iterates on 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;

An iteration loop can be finally built by using the public methods

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 following code fragment 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
```