ISTITUTO
DI
ANALISI NUMERICA

del

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

PAVIA
1999

PUBBLICAZIONI

N. 1166

Enrico Bertolazzi, Gianmarco Manzini

The Kernel of P2MESH

The Kernel of P2MESH

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 some background details about the library classes and their implementation.

(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 Introduction

One of the major difficulties in developing a PDE solver relies in the manipulation of topological and geometrical data. Very surprisingly, few attention is usually payed by developers of numerical algorithms to the way this kind of information is represented in a real application program. Mesh representation is generally considered the business of developers of mesh generation algorithms, while numericians tend to consider this issue as a minor implementation detail.

Much of the current programming practice in scientific computing is actually "ad hoc" and basically procedural. This fact historically happens because for the last decades the main concern has largely been the one of efficiency. A numerical simulation code usually repeats many thousands (or millions) of times the same sequence of actions, identically applied to the most elementary entities of the numerical discretization involved, no matter these latter ones being control volumes, edges, points, or others.

It was only a few years ago that programming techniques could be summarized as just the optimization of loops by unrolling, vectorization and more recently by parallelizations. Nowadays, some of these latter tasks can be taken into account in a number of situations by compilers especially for imperative or procedural languages such as FORTRAN or C.

In modern software development, reliability, correctness, maintenance and re-usability are considered issues of primary interest. Hence, there is a demand for a more sophisticated code structuring, such as the ones provided by non-procedural paradigms.

On the other hand, the standard approach in the existing software packages, either the ones in commercial distributions or the ones available in public domain or freeware circuits, consists in furnishing a set of predefined problem-solver black-boxes. In this spirit, several full contained packages have been proposed in the last years to the scientific community, see for example KASKADE[1], DIFFPACK[2], UG[3], etc.

This kind of implementations asks the final user to select the tool in the library which is most appropriate to the problem at hand. Even if this way might allow a very cheap and straightforward solution to most common – although complex – applications, it is indeed very limiting. The development of a numerical solver for non-standard research applications may be very difficult, either in the case of academic or industrial

[1] ftp://elib.zib-berlin.de/pub/kaskade
[2] http://www.nobjects.com/Diffpack/
[3] http://www.ica3.uni-stuttgart.de/~ug

problems. The final software product is then likely to get a very hard and expensive development and actually be unsatisfactory from the viewpoint of efficiency, re-usability, and maintenance.

The P2MESH software system is based on a substantially different approach. The kernel library is a set of template definitions: since the internal implementation of the library class templates are hidden to the final end user, all data set are accessible only through suitable public interfaces. This feature implements the crucial idea of data hiding. Some well known drawbacks may characterize a software, whose basic data structures are not protected, and are thus freely accessible by any part of the program. A local modification may be the source of subtle bugs and may demand for a massive series of global checks in order to verify its correctness.

The P2MESH software system development has been carried out by taking care of all these issues. The library consists in a single header file which must be included in the user programs, hence, no library object files must be linked and no particular installation procedure is required. The P2MESH software system must be compiled by a C++ compiler that accepts a subset of the template specifications detailed in the ISO C++ language [4].

2 The kernel implementation

The ease of implementation and the successive performances of a program based on P2MESH are strongly affected by the design of the kernel. As shown in figure 1, user-defined data structures inherit the generic kernel definitions and then specializes the by introducing details and specifications of the computational problem at hand.

A template class is indeed able to store the information needed for the allocation of any user-defined objects instantiated in the code and any functionalities to manipulate it, and should then serve as an administrator that delegates the actual work to specific functions. In this way, it is ensured that computations will benefit from the detailed knowledge of each data type.

The strategic choice concerns with what should be common to all final applications and what should instead be supplied time-by-time by the end user.

[4] The egcs compiler freely distributed for example at http://egcs.cygnus.com fulfills this requirement.

It is worth noting that the present approach makes an extensive usage of static polymorphism, inheritance and template technology supported by C++, while it does not implement some other typical C++ ingredients such as virtual functions and abstract classes, due to performance considerations.

The P2MESH operates on user defined classes that are unknown to the library. For example the methods that returns the first vertex of an edge class do not known the details of the edge class until they are specified by the user.

To circumvent this difficulty some specific C++ template techniques are employed [2, 3]. The P2MESH library is builded using this methodology so that the resulting code is flexible without loss of efficiency. However, these techniques are transparent to the final user.

This kernel design is, perhaps, not the *unique* or the *best* one in absolute, but it has been experienced as *good enough* because it allows end-user implementations which are at the same time simple and computational efficient.

A brief tutorial survey of the way template-based generic programming is used in P2MESH is given in the rest of the section. The base classes of a hypothetical library are indicated with the names `GenericVertex`, `GenericEdge` and `GenericTriangle`. These names are fictitious and chosen just for the presentation, and do not correspond to the names of the types adopted in the real implementation.

Let us suppose that one wants to write an algorithm that operates on the instances of type “vertex”, “edge” and “triangle” of a mesh. Let us also suppose that such algorithm can be completely defined once the topological information about the mesh are known, which implies that a “generic” implementation is possible, before an “extended” definition is provided via public inheritance in the user application program. An eventual generic definition is illustrated in the following example

```
1 // file: ex1.hh
2 class GenericVertex { } ;

3 class GenericEdge {
4   GenericVertex * pv[2] ;
5 public:
6   GenericVertex & v(unsigned i) { return * pv[i] ; }
7   void set(GenericVertex & v0, GenericVertex & v1)
8     { pv[0] = &v0 ; pv[1] = &v1 ; }
9 } ;
```

```
10 class GenericTriangle {
11   GenericEdge * pe[3] ;
12 public:
13   GenericEdge & e(unsigned i) { return * pe[i] ; }
14   void set(GenericEdge & e0, GenericEdge & e1, GenericEdge & e2)
15     { pe[0] = &e0 ; pe[1] = &e1 ; pe[2] = &e2 ; }
16 } ;
```

The following piece of code would naively use the definitions introduced in the previous header file, but it can not be compiled.

```
1  // file: ex1.cc
2  # include "ex1.hh"

3  class Vertex   : public GenericVertex {} ;
4  class Edge     : public GenericEdge {} ;
5  class Triangle : public GenericTriangle {} ;

6  int
7  main() {
8    Vertex   v0, v1, v2 ;
9    Edge     e0, e1, e2 ;
10   Triangle t ;
11   // initialize edges
12   e0.set(v0,v1) ; e1.set(v1,v2) ; e2.set(v0,v2) ;
13   // initialize triangle
14   t.set(e0, e1, e2) ;
15   // The casting is necessary because the type
16   // GenericVertex cannot be automatically converted
17   // to the type Edge;
18   Edge & re = static_cast<Edge&>(t . e (0)) ;
19 } ;
```

The compilation fails because the assignment in line `19` requires a conversion form the type `GenericEdge` returned by `GenericTriangle::e(unsigned i)` to the expected type `Edge`. A conversion operator is not explicitly provided in the definition of the class `Edge` and the hypothetical library class `GenericTriangle` can not know anything about the project class `Edge`. A very rough solution would consist in casting the returned type to the correct one, but casting operations are usually discouraged because they prevent compiler type controls and reduce software reliability [1]. This problem of implementation can be circumvented in C++ by using a special template programming technique.

```
// file: ex2.hh
class GenericVertex { } ;

template <typename V>
class GenericEdge {
  V * pv[2] ;
public:
  V & v(unsigned i) { return * pv[i] ; }
  void set(V & v0, V & v1) { pv[0] = &v0 ; pv[1] = &v1 ; }
} ;

template <typename V, typename E>
class GenericTriangle {
  E * pe[3] ;
public:
  E & e(unsigned i) { return * pe[i] ; }
  void set(E & e0, E & e1, E & e2)
    { pe[0] = &e0 ; pe[1] = &e1 ; pe[2] = &e2 ; }
} ;
```

Notice that the generic class definitions are parametrized by user-defined types known at the moment of the instantiation phase. This technique is derived from [2] and is illustrated in the following piece of code.

```
// file: ex2.cc
# include "ex2.hh"

class Vertex   : public GenericVertex {} ;
class Edge     : public GenericEdge<Vertex> {} ;
class Triangle : public GenericTriangle<Vertex,Edge> {} ;

int
main() {
  Vertex   v0, v1, v2 ;
  Edge     e0, e1, e2 ;
  Triangle t ;
  // initialize edges
  e0.set(v0,v1) ; e1.set(v1,v2) ; e2.set(v0,v2) ;
  // initialize triangle
  t.set(e0, e1, e2) ;
  // now method is safely inherited
  Edge & re = t . e (0) ;
} ;
```

Since the type name `Edge` is statically introduced in the parameter argument list of `GenericTriangle` the member function `GenericTriangle::e(unsigned i)`, inherited by `Triangle`, is able to return the correct type. This rather tricky technique works because the standard ANSI C++ allows the instantiation of a class which has been partially defined. Indeed, the methods of a class are instantiated only when they are called by the program. Notice that no casting is required.

This implementation suffers however of an excessive number of parameters and the complexity grows up with the number of classes involved. A better organization is possible as follows.

```
1  // file ex3.hh
2  template <typename V_type, typename E_type, typename T_type>
3  class GenericCommon {
4  public:
5    typedef V_type V ;
6    typedef E_type E ;
7    typedef T_type T ;
8  } ;

9  template <typename COMMON> class GenericVertex {
10 public:
11   typedef typename COMMON::V V ;
12   typedef typename COMMON::E E ;
13   typedef typename COMMON::T T ;
14 } ;

15 template <typename COMMON> class GenericEdge {
16 public:
17   typedef typename COMMON::V V ;
18   typedef typename COMMON::E E ;
19   typedef typename COMMON::T T ;
20   V & v(unsigned i) { return * pv[i] ; }
21   void set(V & v0, V & v1) { pv[0] = &v0 ; pv[1] = &v1 ; }
22 private:
23   V * pv[2] ;
24 } ;

25 template <typename COMMON> class GenericTriangle {
26 public:
27   typedef typename COMMON::V V ;
28   typedef typename COMMON::E E ;
29   typedef typename COMMON::T T ;
```

```
30   E & e(unsigned i) { return * pe[i] ; }
31   void set(E & e0, E & e1, E & e2)
32     { pe[0] = &e0 ; pe[1] = &e1 ; pe[2] = &e2 ; }
33 private:
34   E * pe[3] ;
35 } ;
```

In this code a pivotal class `GenericCommon` is used to store the user data type. In this way the complexity of the templates is confined to the pivotal class `GenericCommon`.

```
1  // file: ex3.cc
2  # include "ex3.hh"

3  class Vertex ;
4  class Edge ;
5  class Triangle ;

6  class Common   : public GenericCommon<Vertex,Edge,Triangle> {} ;
7  class Vertex   : public GenericVertex<Common> {} ;
8  class Edge     : public GenericEdge<Common> {} ;
9  class Triangle : public GenericTriangle<Common> {} ;

10 int
11 main() {
12   Vertex   v0, v1, v2 ;
13   Edge     e0, e1, e2 ;
14   Triangle t ;
15   // initialize edges
16   e0.set(v0,v1) ; e1.set(v1,v2) ; e2.set(v0,v2) ;
17   // initialize triangle
18   t.set(e0, e1, e2) ;
19   Edge & re = t.e (0) ;
20 } ;
```

The generic classes are extended by introducing user member data and functions as shown in the example.

```
1  // file: ex4.cc
2  # include "ex3.hh"

3  // User Specialization
```

```
class Vertex ;
class Edge ;
class Triangle ;

class Common : public GenericCommon<Vertex,Edge,Triangle> {} ;

class Vertex : public GenericVertex<Common> {
public:
  double x, y ;
} ;

class Edge : public GenericEdge<Common> {
public:
  double nx() { return v(1).x - v(0).x ; }
  double ny() { return v(1).y - v(0).y ; }
} ;

class Triangle : public GenericTriangle<Common> {} ;

int
main() {
  Vertex   v0, v1, v2 ;
  Edge     e0, e1, e2 ;
  Triangle t ;
  // initialize vertices
  v0.x = 0 ; v0.y = 0 ; v1.x = 1 ; v1.y = 1 ; v2.x = 0 ; v2.y = 1 ;
  // initialize edges
  e0.set(v0,v1) ; e1.set(v1,v2) ; e2.set(v0,v2) ;
  // initialize triangle
  t.set(e0, e1, e2) ;
  // use added functionality
  double nx = e0 . nx() ;
  double ny = e0 . ny() ;
} ;
```

A highly modular programming environment can be easily designed by using extensively the “curiously template recursive pattern” also known as *Furnish Trick*,[4]

```
// file: ex5.cc
# include "ex3.hh"

// Additional functionality
template <typename COMMON>
class AdditionalTriangle {
```

```
public:
  typedef typename COMMON::V V ;
  typedef typename COMMON::E E ;
  typedef typename COMMON::T T ;
  V & v(unsigned i) {
    E & E0 = leaf() . e(i) ;
    E & E1 = leaf() . e((i+1)%3) ;
    unsigned j = (&E0.v(1) == &E1.v(0) || &E0.v(1) == &E1.v(1)) ?0:1;
    return E0 . v(j) ;
  }
private:
  T & leaf(void) { return static_cast<T&>(*this) ; }
} ;

// User Specialization

class Vertex ;
class Edge ;
class Triangle ;

class Common : public GenericCommon<Vertex,Edge,Triangle> {} ;

class Vertex : public GenericVertex<Common> {
public:
  double x, y ;
} ;

class Edge : public GenericEdge<Common> {
public:
  double nx() { return v(1).x - v(0).x ; }
  double ny() { return v(1).y - v(0).y ; }
} ;

class Triangle : virtual public GenericTriangle<Common>,
                 public AdditionalTriangle<Common> {} ;

int
main() {
  Vertex   v0, v1, v2 ;
  Edge     e0, e1, e2 ;
  Triangle t ;
  // initialize edges
  e0.set(v0,v1) ; e1.set(v1,v2) ; e2.set(v0,v2) ;
  // initialize triangle
  t.set(e0, e1, e2) ;
  // use added functionality
```

```
45   Vertex & v = t.v(2) ;
46 } ;
```

Exploiting this recursive pattern, it is possible to parameterize both the generic base classes and some other new additional ones. Multiple public inheritance allows the derived application classes to access to both basic and additional methods, see lines 32–33.

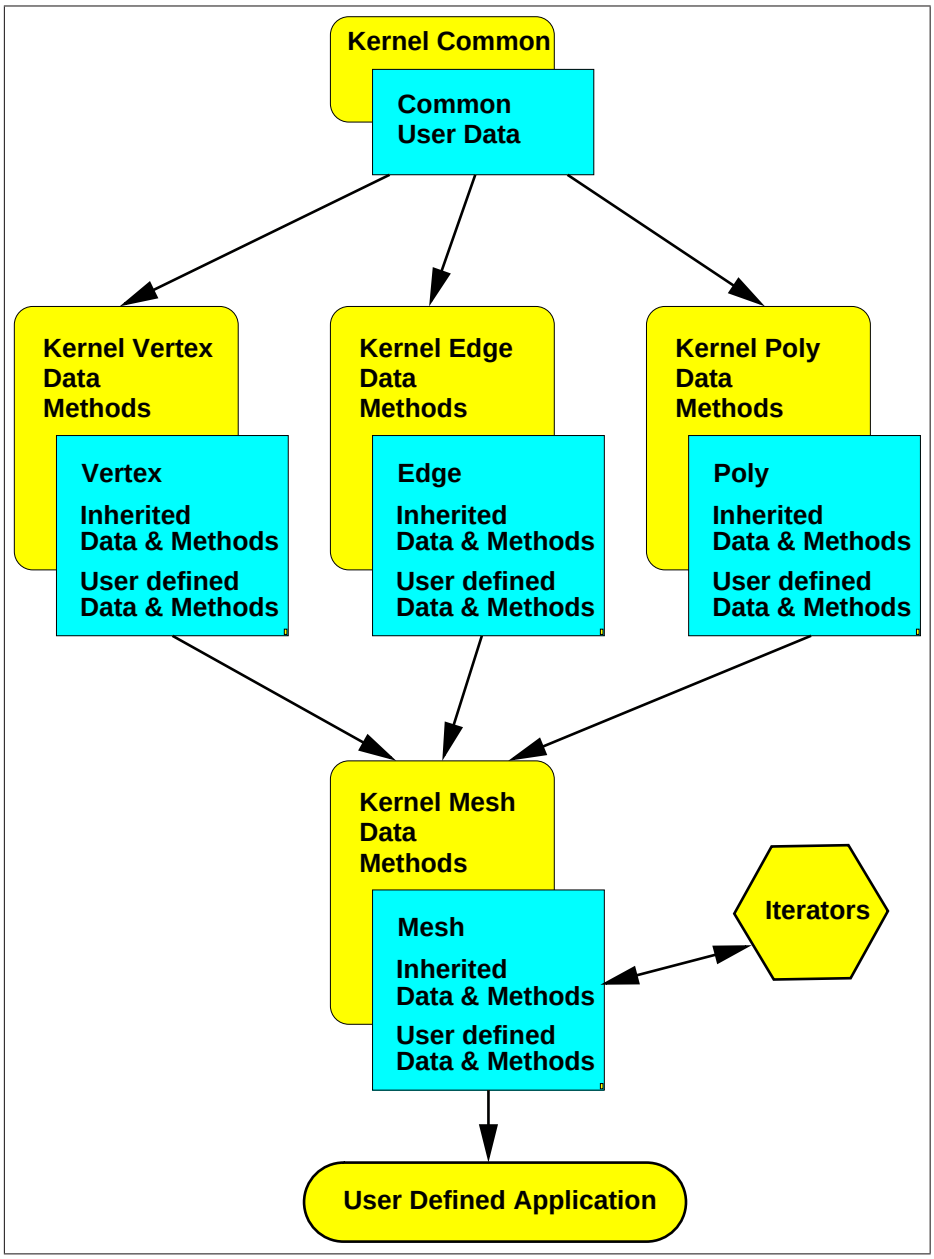

Figure 1: An application program supported by a parametric mesh-manager

3 Description of the kernel

3.1 The base class p2_common.

```
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 {
public:

  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 ;

  static unsigned const Size = SIZE_value ;
  static unsigned const List = LIST_value ? 1 : 0 ;

  // internal stuff
} ;
```

Description

`p2_common` holds the static information which is shared by all the objects instantiated by the library based programs at run-time.

Public Types

Public types are aliased via a `typedef` definition and are inherited by the user defined classes.

The class contains the typedef aliases for the user-defined names of the classes derived by `p2_vertex`, `p2_edge`, `p2_poly` and `p2_mesh`. This issue allows the template parameterization of the library by the user class names. No default is given because the project class name specification is mandatory.

P2V typedef alias for the user defined class publicly derived from the library class `p2_vertex`;

P2E typedef alias for the user defined class publicly derived from the library class `p2_edge`;

P2P typedef alias for the user defined class publicly derived from the library class `p2_poly`;

P2M typedef alias for the user defined class publicly derived from the library class `p2_mesh`;

The class holds the definition for the types of signed integer, unsigned integer and real numbers, which, by default, are respectively `int`, `unsigned` and `double` type.

The default types are automatically selected by the library if not otherwise specified by the user. For example, the user may define `long int` instead of `int`, `unsigned long` instead of `unsigned`, `float`, `long double` instead of `double`. This feature makes possible the usage of numerical types for higher precision arithmetic.

Integer typedef alias for signed integer numbers, default type: `int`;

Unsigned typedef alias for unsigned integer numbers, default type: `unsigned`;

Real typedef alias for real numbers, default type: `double`;

Vmark typedef alias for vertex markers, default type: `unsigned`;

Emark typedef alias for edge markers, default type: `unsigned`;

Pmark typedef alias for polygon markers, default type: `unsigned`;

Public Constants

The class contains also the static size of the polygon instances, i.e. the number of vertices or edges in the polygon vertex- and edge-list.

`Size` constant of type `unsigned`; its value is `3` (default) for a triangular mesh and `4` for a quadrilateral mesh;

`List` constant flag of type `bool` (boolean); its value is `true` if the instances of the class `p2_vertex` will contain additional lists of connected vertices, edges, and polygons, otherwise it is `false` (default).

3.2 The base class `p2_vertex`.

```
template <typename P2_COMMON>
class p2_vertex : private p2_vertex_variant<P2_COMMON::List>,
                  public P2_COMMON {

  typedef typename P2_COMMON::Real     Real ;
  typedef typename P2_COMMON::Unsigned Unsigned ;
  typedef typename P2_COMMON::Integer  Integer ;
  typedef typename P2_COMMON::P2V      P2V ;
  typedef typename P2_COMMON::P2E      P2E ;
  typedef typename P2_COMMON::P2P      P2P ;
  typedef typename P2_COMMON::P2M      P2M ;

private:
  Real p2_x, p2_y ;

public:
  // access vertex
  Real const & x(void) const ;
  Real const & y(void) const ;
  Real       & x(void) ;
  Real       & y(void) ;

  Unsigned n_vertex(void) const ;
  Unsigned n_edge  (void) const ;
  Unsigned n_poly  (void) const ;

  // access surrounding entities
  P2V const & vertex(Unsigned const nv) const ;
  P2V       & vertex(Unsigned const nv) ;
  P2E const & edge  (Unsigned const ne) const ;
  P2E       & edge  (Unsigned const ne) ;
  P2P const & poly  (Unsigned const np) const ;
  P2P       & poly  (Unsigned const np) ;

  // getting local numbering
  Unsigned local_number(P2V const & rV) const ;
  Unsigned local_number(P2E const & rE) const ;
  Unsigned local_number(P2P const & rP) const ;
} ;
```

Description

`p2_vertex` is the base class for the user defined vertex class. The class is publicly derived from the class template `p2_vertex_variant`, which is specified at compile-time by the boolean flag `List` in `p2_common`. The two different versions of the class `p2_vertex_variant` are used to implement the base vertex class with and without the list of the connected mesh entities. No further details are given about the implementation of this issue hereafter, whilst the usage is illustrated by several examples in the chapter on the public interface.

Inherited Types (see the description for the class `p2_common`).

- `Integer`
- `Unsigned`
- `Real`
- `P2V`
- `P2E`
- `P2P`
- `P2M`
- `Vmark`
- `Emark`
- `Pmark`

Topological Methods

The following methods are active only if the boolean flag `List` is `true` at compile-time; when they are invoked with `List=false`, they produce a run-time error;

n_vertex `Unsigned n_vertex(void) const`

returns the number of vertices connected to the current vertex instance;

vertex
```
P2V const & vertex(Unsigned const nv) const
P2V       & vertex(Unsigned const nv)
```
returns the reference to the `nv`-th vertex in the list of the connected vertices;

n_edge
```
Unsigned n_edge(void) const
```
returns the number of edges incident to the current vertex instance;

edge
```
P2E const & edge(Unsigned const ne) const
P2E       & edge(Unsigned const ne)
```
returns the reference to the `ne`-th edge in the list of the incident edges;

n_poly
```
Unsigned n_poly(void) const
```
returns the number of polygons incident to the current vertex instance;

poly
```
P2P const & poly(Unsigned const np) const
P2P       & poly(Unsigned const np)
```
returns the reference to the `np`-th polygon in the list of the incident polygons;

Geometrical Methods

---

x
```
Real const & x(void) const
Real       & x(void)
```
returns a reference to the coordinate `x` of the current vertex instance;

y
```
Real const & y(void) const
Real       & y(void)
```
returns a reference to the coordinate `y` of the current vertex instance.

3.3 The base class p2_edge.

```
template <typename P2_COMMON>
class p2_edge : public P2_COMMON {

  typedef typename P2_COMMON::Real     Real ;
  typedef typename P2_COMMON::Unsigned Unsigned ;
  typedef typename P2_COMMON::Integer  Integer ;
  typedef typename P2_COMMON::P2V      P2V ;
  typedef typename P2_COMMON::P2E      P2E ;
  typedef typename P2_COMMON::P2P      P2P ;
  typedef typename P2_COMMON::P2M      P2M ;

private:
  P2P * p2_p[2] ;
  P2V * p2_v[2] ;

public:
  Unsigned n_vertex(void) const ;
  Unsigned n_edge  (void) const ;
  Unsigned n_poly  (void) const ;

  // accessing surrounding entities
  P2V const & vertex(Unsigned const nv) const ;
  P2V       & vertex(Unsigned const nv) ;
  P2E const & edge  (Unsigned const ne) const ;
  P2E       & edge  (Unsigned const ne) ;
  P2P const & poly  (Unsigned const np) const ;
  P2P       & poly  (Unsigned const np) ;

  // return ‘‘true’’ if the polygon np exists
  bool ok_poly(Unsigned const np) const ;

  // getting local numbering
  Unsigned local_number(P2V const & rV) const ;
  Unsigned local_number(P2E const & rE) const ;
  Unsigned local_number(P2P const & rP) const ;

  // accessing vertex coodinate
  Real const & x(Unsigned const nv) const ;
  Real const & y(Unsigned const nv) const ;

  // accessing midpoint
  Real xm(void) const ;
  Real ym(void) const ;
```

```
  // accessing point on edge at parametric coordinate
  Real xt(Real const & t) const ;
  Real yt(Real const & t) const ;

  // accessing normal, tangential and length
  Real nx(void) const ;
  Real ny(void) const ;
  Real tx(void) const ;
  Real ty(void) const ;
  Real length(void) const ;
} ;
```

Description

p2_edge is the base class for the user defined edge class.

Inherited Types (see the description for the class p2_common)

- Integer
- Unsigned
- Real
- P2V
- P2E
- P2P
- P2M
- Vmark
- Emark
- Pmark

Topological Methods

vertex
```
P2V const & vertex(Unsigned const nv) const
P2V       & vertex(Unsigned const nv)
```

returns the reference to the `nv`-th connected vertex; the edge is assumed oriented from vertex `nv=0` to vertex `nv=1`.

edge
```
P2E const & edge(Unsigned const ne) const
P2E       & edge(Unsigned const ne)
```

returns the reference to the `ne`-th connected edge; the connected edges are counter-clockwise numbered starting from the "left" adjacent polygon;

poly
```
P2P const & poly(Unsigned const np) const
P2P       & poly(Unsigned const np)
```

returns the reference to the `np`-th adjacent polygon; the edge is assumed oriented in such a way that `nv=0` is the left-side polygon and `nv=1` is the right-side one;

ok_poly
```
bool ok_poly(Unsigned const np) const
```

returns `true` if the `np`-th entry for the connected polygon is a reference to an existing mesh polygon, i.e. the edge is not located on the mesh boundary; returns `false` otherwise;

Geometrical Methods

x
```
Real const & x(Unsigned const nv) const
```

returns the coordinate `x` of the connected vertex `nv`, i.e. `vertex(nv).x()`;

y
```
Real const & y(Unsigned const nv) const
```

returns the coordinate `y` of the connected vertex `nv`, i.e. `vertex(nv).y()`;

xm
```
Real xm(void) const
```

returns the coordinate `x` of the edge midpoint, i.e. `0.5*(x(0)+x(1))`;

ym
```
Real ym(void) const
```

returns the coordinate `y` of the edge midpoint, i.e. `0.5*(y(0)+y(1))`;

xt `Real xt(Real const & t) const`

returns the coordinate `x` of the linearly interpolated point at position `t` on the edge, i.e `(1-t)*x(0)+t*x(1);`

yt `Real yt(Real const & t) const`

returns the coordinate `y` of the linearly interpolated point at position `t` on the edge, i.e `(1-t)*x(0)+t*x(1);`

nx `Real nx(void) const`

returns the first component of the vector orthogonal to the edge, with Euclidean norm equal to the length of the edge, i.e. `y(1)-y(0);`

ny `Real ny(void) const`

returns the second component of the vector orthogonal to the edge, with Euclidean norm equal to the length of the edge, i.e. `x(0)-x(1);`

tx `Real tx(void) const`

returns the first component of the vector parallell to the edge, with Euclidean norm equal to the length of the edge, i.e. `x(1)-x(0);`

ty `Real ty(void) const`

returns the second component of the vector parallel to the edge, with Euclidean norm equal to the length of the edge, i.e. `y(1)-y(0);`

length `Real length(void) const`

returns the length of the edge; i.e. $\sqrt{(\texttt{x(1)-x(0)})^2 + (\texttt{y(1)-y(0)})^2}$.

3.4 The base class p2_poly.

```
template <typename P2_COMMON>
class p2_poly : public P2_COMMON {

  typedef typename P2_COMMON::Real     Real ;
  typedef typename P2_COMMON::Unsigned Unsigned ;
  typedef typename P2_COMMON::Integer  Integer ;
  typedef typename P2_COMMON::P2V      P2V ;
  typedef typename P2_COMMON::P2E      P2E ;
  typedef typename P2_COMMON::P2P      P2P ;
  typedef typename P2_COMMON::P2M      P2M ;

private:
  P2E * p2_e [Size] ;
  P2V * p2_v [Size] ;

public:
  Unsigned n_vertex(void) const ;
  Unsigned n_edge  (void) const ;
  Unsigned n_poly  (void) const ;

  // accessing surrounding entities
  P2V const & vertex(Unsigned const nv) const ;
  P2V       & vertex(Unsigned const nv) ;
  P2E const & edge  (Unsigned const ne) const ;
  P2E       & edge  (Unsigned const ne) ;
  P2P const & poly  (Unsigned const np) const ;
  P2P       & poly  (Unsigned const np) ;

  // return ‘‘true’’ if the polygon np exists
  bool ok_poly(Unsigned const np) const ;

  // getting local numbering
  Unsigned local_number(P2V const & rV) const ;
  Unsigned local_number(P2E const & rE) const ;
  Unsigned local_number(P2P const & rP) const ;

  // attibute of the poly
  bool ok_oriented(Unsigned const ne) const ;

  // the coordinate of the vertex
  Real const & x(Unsigned const nv) const ;
  Real const & y(Unsigned const nv) const ;
```

```
  // the centroid of the poly
  Real xc(void) const ;
  Real yc(void) const ;

  // the area of the poly
  Real area(void) const ;

  // outward normals
  Real nx(Unsigned const ne) const ;
  Real ny(Unsigned const ne) const ;

  // counterclockwise tangential
  Real tx(Unsigned const ne) const ;
  Real ty(Unsigned const ne) const ;

  // the midpoints of the edges
  Real xm(Unsigned const ne) const ;
  Real ym(Unsigned const ne) const ;

  // the point at coordinate t on the i edge
  Real xt(Unsigned const ne, Real const & t) const ;
  Real yt(Unsigned const ne, Real const & t) const ;
  Real length(Unsigned const ne) const ;

  // special function for triangles & rectangles
  void jacobian(Real const & s, Real const & t, Real J[2][2]) const;
  void inverse_jacobian(Real const & s, Real const & t,
                        Real J[2][2]) const ;

  // transform s,t coordinate to x,y real coordinate
  void st_to_xy(Real const & s, Real const & t,
                Real       & x, Real       & y) const ;

  // transform x, y coordinate to s, t local coordinate
  void xy_to_st(Real const & x, Real const & y,
                Real       & s, Real       & t) const ;

  // transform grad s,t to grad x,y
  void grad_st_to_xy(Real const & s,      Real const & t,
                     Real const   gst[2], Real gxy[2]) const ;

  // transform grad s,t to grad x,y
  void grad_xy_to_st(Real const & x,    Real const & y,
                     Real const gxy[2], Real gst[2]) const ;
} ;
```

Description

`p2_poly` is the base class for the user-defined polygon class. The methods hereafter, if not otherwise indicated, refers to the features of a given instance of polygon type, the *current polygon instance*.

Inherited Types (see the description for the `p2_common` class)

- `Integer`
- `Unsigned`
- `Real`
- `P2V`
- `P2E`
- `P2P`
- `P2M`
- `Vmark`
- `Emark`
- `Pmark`

Topological Methods

n_vertex `Unsigned n_vertex(void) const`

returns the number of vertices of the polygon, which is `3` for a triangular mesh and `4` for a quadrilateral one;

vertex
```
P2V const & vertex(Unsigned const nv) const
P2V       & vertex(Unsigned const nv)
```

returns the reference to the `nv`-th vertex in the polygon; vertices are assumed to be numbered counterclockwise starting from `nv=0`;

n_edge `Unsigned n_edge(void) const`

returns the number of edges in the polygon, which is `3` for a triangular mesh and `4` for a quadrilateral one;

edge
```
P2E const & edge(Unsigned const ne) const
P2E       & edge(Unsigned const ne)
```

returns the reference to the `ne`-th edge in the polygon; edges are assumed to be numbered counterclockwise starting from `ne=0`;

poly
```
P2P const & poly(Unsigned const np) const
P2P       & poly(Unsigned const np)
```

returns the reference to the `np`-th adjacent polygon to the current polygon instance;

ok_poly `bool ok_poly(Unsigned const np) const`

returns `true` if there exists a polygon in the mesh polygon list sharing the `np`-th edge;

ok_oriented `bool ok_oriented(Unsigned const ne) const`

local_number `Unsigned local_number(P2V const & rV) const`

returns the local position of the vertex referenced by `rV` in the vertex list of the polygon; if `rV` is not in the vertex list a run-time error is produced;

`Unsigned local_number(P2E const & rE) const`

returns the local position of the edge referenced by `rE` in the edge list of the polygon; if `rE` is not in the edge list a run-time error is produced;

`Unsigned local_number(P2P const & rP) const`

returns the local position of the edge shared by the polygon referenced by `rP` in the edge list of the current polygon instance; if `rP` is not an adjacent polygon a run-time error is produced.

Geometrical Methods

x `Real const & x(Unsigned const nv) const`

returns the coordinate `x` of the vertex `nv` in the vertex list of the current polygon instance;

y `Real const & y(Unsigned const nv) const`

returns the coordinate `y` of the vertex `nv` in the vertex list of the current polygon instance;

xc `Real xc(void) const`

returns the coordinate `x` of the centroid of the vertices of the current polygon instance, defined as

$$\texttt{x()} = \frac{1}{n}\sum_{i=0}^{n}\texttt{x(i)}$$

where $n =$`n_vertex()`

yc `Real yc(void) const`

returns the coordinate `y` of the centroid of the vertices of the current polygon instance, defined as

$$\texttt{y()} = \frac{1}{n}\sum_{i=0}^{n}\texttt{y(i)}$$

where $n =$`n_vertex()`

area `Real area(void) const`

returns the area of the polygon;

nx `Real nx(Unsigned const ne) const`

returns the first component of the vector orthogonal to the edge `ne`; the orthogonal vector is assumed to be oriented outward the polygon and its Euclidean norm is equal to the length of the specified edge;

ny `Real ny(Unsigned const ne) const`

returns the second component of the vector orthogonal to the edge `ne`; the orthogonal vector is assumed to be oriented outward the polygon and its Euclidean norm is equal to the length of the specified edge;

tx `Real tx(Unsigned const ne) const`

returns the first component of the vector parallel to the edge `ne`; the vector is assumed to be oriented counterclockwise the polygon and its Euclidean norm is equal to the length of the specified edge;

ty `Real ty(Unsigned const ne) const`

returns the second component of the vector parallel to the edge `ne`; the orthogonal vector is assumed to be oriented counterclockwise the polygon and its Euclidean norm is equal to the length of the specified edge;

xm `Real xm(Unsigned const ne) const`

returns the coordinate `x` of the midpoint of the edge `ne` in the polygon, i.e.

```
edge(ne).xm();
```

ym `Real ym(Unsigned const ne) const`

returns the coordinate `y` of the midpoint of the edge `ne` in the polygon, i.e.

```
edge(ne).ym();
```

xt `Real xt(Unsigned const ne, Real const & t) const`

returns the coordinate `x` of the linearly interpolated point at position `t` on the edge `ne`, i.e `(1-t)*x(ne)+t*x(ne+1)`; the method is **not** equivalent to `edge(ne).xt(t)`, because the edge orientation may be different, i.e. equivalence is given only when the edge of the polygon has the same orientation of the edge as instance of `p2_edge`.

yt `Real yt(Unsigned const ne, Real const & t) const`

returns the coordinate `y` of the linearly interpolated point at position `t` on the edge `ne`, i.e `(1-t)*y(ne)+t*y(ne+1)`; the method is **not** equivalent to `edge(ne).yt(t)`, because the edge orientation may be different, i.e. equivalence is given only when the edge of the polygon has the same orientation of the edge as instance of `p2_edge`.

3.5 The base class p2_mesh.

```
template <typename P2_COMMON>
class p2_mesh : public P2_COMMON {

  typedef typename P2_COMMON::Real     Real ;
  typedef typename P2_COMMON::Unsigned Unsigned ;
  typedef typename P2_COMMON::Integer  Integer ;

  typedef typename P2_COMMON::P2V      P2V ;
  typedef typename P2_COMMON::P2E      P2E ;
  typedef typename P2_COMMON::P2P      P2P ;
  typedef typename P2_COMMON::P2M      P2M ;

  typedef typename P2_COMMON::Vmark    Vmark ;
  typedef typename P2_COMMON::Emark    Emark ;
  typedef typename P2_COMMON::Pmark    Pmark ;

  typedef vector<P2V> P2_LIST_V ;
  typedef vector<P2E> P2_LIST_E ;
  typedef vector<P2P> P2_LIST_P ;

public:
  typedef typename P2_LIST_V::iterator       vertex_iterator ;
  typedef typename P2_LIST_V::const_iterator vertex_const_iterator ;

  typedef typename P2_LIST_E::iterator       edge_iterator ;
  typedef typename P2_LIST_E::const_iterator edge_const_iterator ;

  typedef typename P2_LIST_P::iterator       poly_iterator ;
  typedef typename P2_LIST_P::const_iterator poly_const_iterator ;

private:
  Unsigned p2_n_vertex  ; // total numbers of vertices
  Unsigned p2_n_bvertex ; // boundary vertices
  Unsigned p2_n_ivertex ; // internal vertices
  P2_LIST_V p2_vlist ;

  Unsigned p2_n_edge    ; // total numbers of edges
  Unsigned p2_n_bedge   ; // boundary edges
  Unsigned p2_n_iedge   ; // internal edges
  P2_LIST_E p2_elist ;

  Unsigned p2_n_poly    ; // total numbers of polys
  Unsigned p2_n_bpoly   ; // boundary polys
```

```
  Unsigned p2_n_ipoly   ; // internal polys
  P2_LIST_P p2_plist ;

  void JointEdges(void) ;
  void BuildEdges(void) ;
  void Reorder(void) ;
  void ReorderList(void) ;

public:

  void bbox(Real & xmin, Real & ymin,
            Real & xmax, Real & ymax) const ;

  Unsigned const & n_vertex (void) const { return p2_n_vertex ; }
  Unsigned const & n_bvertex(void) const { return p2_n_bvertex ; }
  Unsigned const & n_ivertex(void) const { return p2_n_ivertex ; }
  Unsigned const & n_edge   (void) const { return p2_n_edge ; }
  Unsigned const & n_bedge  (void) const { return p2_n_bedge ; }
  Unsigned const & n_iedge  (void) const { return p2_n_iedge ; }
  Unsigned const & n_poly   (void) const { return p2_n_poly ; }
  Unsigned const & n_bpoly  (void) const { return p2_n_bpoly ; }
  Unsigned const & n_ipoly  (void) const { return p2_n_ipoly ; }

  P2V const & vertex(Unsigned const nv) const ;
  P2V       & vertex(Unsigned const nv) ;
  P2E const & edge  (Unsigned const ne) const ;
  P2E       & edge  (Unsigned const ne) ;
  P2P const & poly  (Unsigned const np) const ;
  P2P       & poly  (Unsigned const np) ;

  Unsigned local_number(P2V const & rV) const ;
  Unsigned local_number(P2E const & rE) const ;
  Unsigned local_number(P2P const & rP) const ;

  // iterators definitions........
  // ........
  // mesh builder definitions
  // ........
  // print mesh stuff
  // ...........
} ;
```

Description

`p2_mesh` is the base class for the user defined mesh class.

Inherited Types (see the description for the `p2_common` class)

- `Integer`
- `Unsigned`
- `Real`
- `P2V`
- `P2E`
- `P2P`
- `P2M`
- `Vmark`
- `Emark`
- `Pmark`

Methods

bbox
```
void bbox(double & xmin,
          double & ymin,
          double & xmax,
          double & ymax)
```
computes the bounding box of the mesh;

n_vertex
```
Unsigned n_vertex(void) const
```
returns the number of vertices in the mesh;

n_bvertex
```
Unsigned n_bvertex(void) const
```
returns the number of boundary vertices in the mesh;

n_ivertex `Unsigned n_ivertex(void) const`

returns the number of internal vertices in the mesh;

vertex
```
P2V const & vertex(Unsigned const nv) const
P2V       & vertex(Unsigned const vn)
```

returns the reference to the `nv`-th vertex in the vertex list of the mesh;

n_edge `Unsigned n_edge(void) const`

returns the number of edges in the mesh;

n_bedge `Unsigned n_bedge(void) const`

returns the number of boundary edges in the mesh;

n_iedge `Unsigned n_iedge(void) const`

returns the number of internal edges in the mesh;

edge
```
P2E const & edge(Unsigned const ne) const
P2E       & edge(Unsigned const ne)
```

returns the reference to the `ne`-th edge in the edge list of the mesh;

n_poly `Unsigned n_poly(void) const`

returns the number of polygons in the mesh;

n_bpoly `Unsigned n_bpoly(void) const`

returns the number of boundary polygons in the mesh;

n_ipoly `Unsigned n_ipoly(void) const`

returns the number of internal polygons in the mesh;

poly
```
P2P const & poly(Unsigned const np) const
P2P       & poly(Unsigned const np)
```

returns the reference to the `np`-th polygon in the list of the mesh;

local_number `Unsigned local_number(P2V const & rV) const`

returns the local position (starting form `0`) of the vertex referenced by `rV` in the mesh;

`Unsigned local_number(P2E const & rE) const`

returns the local position (starting form `0`) of the edge referenced by `rE` in the mesh;

```
Unsigned local_number(P2P const & rP) const) const
```

returns the local position (starting form `0`) of the polygon referenced by `rP` in the mesh;

Print mesh stuff

report `void report(ostream & s) const`

reports on the stream `s` some statistics about the mesh.

test_mesh `bool test_mesh(void) const`

performs a severe control on the mesh, producing a run-time error output if a mesh inconsistency is encountered.

print
```
void print(ostream&, Unsigned const =0) const
void print(ostream&, P2E const &, Unsigned const =0) const
void print(ostream&, P2V const &, Unsigned const =0) const
void print(ostream&, P2P const &, Unsigned const =0) const
```

These methods print in a human readable form the mesh data sets. The value of `base` is the offset for indexing, typically `1` for FORTRAN programmers and `0` for C and C++ ones.

3.6 Mesh builders

The following mesh builders are described in details in the programmer's manual. For sake of completeness, their prototypes are reported in this section.

tensor_mesh

```
void
tensor_mesh(
  Real const & xmin, // bounding box of the mesh
```

```
  Real const & xmax,
  Real const & ymin,
  Real const & ymax,
  Unsigned const nx, // x-subdivision
  Unsigned const ny, // y-subdivision
  void (*mark_vertex) ( Vertex &, Vmark const &), // vertex marker
  void (*mark_edge)   ( Edge   &, Emark const &), // edge marker
  void (*mark_poly)   ( Poly   &, Pmark const &), // polygon marker
  Unsigned const kind = 0) // 0 or 1 based index vectors
```

std_tensor_mesh

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

map_mesh

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

read_map_mesh

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

build_mesh

```
void
build_mesh(
  Unsigned const nv,
  Real     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) ;
```

read_mesh

```
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)
```

iterator definitions

From an abstract viewpoint *iterators* are an elegant tool to represent the sequential access of an algorithm to some specified subsets – for example internal and boundary instances of vertex-, edge-, or polygon-type objects stored in the mesh data set. However, the interest in iterators is not only a theoretical one. Their availability in practical implementation ensures, by the data hiding mechanism, the independence of the user application to the actual way mesh data are stored in computer memory and allows an easy way of enhancing computational efficiency in loop cycles. Vertex, edge and polygon lists are implemented in P2MESH by using the STL container `vector`. Iterators are thus available by using the STL ones.

vertex iterators

vertex_begin

```
vertex_iterator       vertex_begin(void)
vertex_const_iterator vertex_begin(void) const
```

these methods return an STL iterator which points to the first element of the vertex list.

ivertex_begin

```
vertex_iterator       ivertex_begin(void)
vertex_const_iterator ivertex_begin(void) const
```

these methods return an STL iterator which points to the first internal element of the vertex list.

bvertex_begin

```
vertex_iterator       bvertex_begin(void)
vertex_const_iterator bvertex_begin(void) const
```

these methods return an STL iterator which points to the first boundary element of the vertex list.

vertex_end

```
vertex_iterator       vertex_end(void)
vertex_const_iterator vertex_end(void) const
```

these methods return an STL iterator which points to the past-to-the-last element of the vertex list.

ivertex_end
```
vertex_iterator       ivertex_end(void)
vertex_const_iterator ivertex_end(void) const
```

these methods return an STL iterator which points to the past-to-the-last internal element of the vertex list.

bvertex_end
```
vertex_iterator       bvertex_end(void)
vertex_const_iterator bvertex_end(void) const
```

these methods return an STL iterator which points to the past-to-the-last boundary element of the vertex list.

edge iterators

edge_begin
```
edge_iterator       edge_begin(void)
edge_const_iterator edge_begin(void) const
```

these methods return an STL iterator which points to the first element of the edge list.

iedge_begin
```
edge_iterator       iedge_begin(void)
edge_const_iterator iedge_begin(void) const
```

these methods return an STL iterator which points to the first internal element of the edge list.

bedge_begin
```
edge_iterator       bedge_begin(void)
edge_const_iterator bedge_begin(void) const
```

these methods return an STL iterator which points to the first boundary element of the edge list.

edge_end
```
edge_iterator       edge_end(void)
edge_const_iterator edge_end(void) const
```

these methods return an STL iterator which points to the past-to-the-last element of the edge list.

iedge_end
```
edge_iterator       iedge_end(void)
edge_const_iterator iedge_end(void) const
```

these methods return an STL iterator which points to the past-to-the-last internal element of the edge list.

bedge_end
```
edge_iterator       bedge_end(void)
edge_const_iterator bedge_end(void) const
```
these methods return an STL iterator which points to the past-to-the-last boundary element of the edge list.

poly iterators

poly_begin
```
poly_iterator       poly_begin(void)
poly_const_iterator poly_begin(void) const
```
these methods return an STL iterator which points to the first element of the polygon list.

ipoly_begin
```
poly_iterator       ipoly_begin(void)
poly_const_iterator ipoly_begin(void) const
```
these methods return an STL iterator which points to the first internal element of the polygon list.

bpoly_begin
```
poly_iterator       bpoly_begin(void)
poly_const_iterator bpoly_begin(void) const
```
these methods return an STL iterator which points to the first boundary element of the polygon list.

poly_end
```
poly_iterator       poly_end(void)
poly_const_iterator poly_end(void) const
```
these methods return an STL iterator which points to the past-to-the-last element of the polygon list.

ipoly_end
```
poly_iterator       ipoly_end(void)
poly_const_iterator ipoly_end(void) const
```
these method return an STL iterator which points to the past-to-the-last internal element of the polygon list.

bpoly_end
```
poly_iterator       bpoly_end(void)
poly_const_iterator bpoly_end(void) const
```
these methods return an STL iterator which points to the past-to-the-last boundary element of the polygon list.

3.7 The class `Iterator`.

```
template <typename P2OBJ>
class Iterator {
  // ... some technical stuff
private:

  P2M * pMesh ;
  vector<P2OBJ>::iterator i ;
  vector<P2OBJ>::iterator start ;
  vector<P2OBJ>::iterator past_end ;

public:

  Iterator(void) ;
  Iterator(P2M & Mesh_, Unsigned const p2loop_ = 0 ) ;
  Iterator(P2M * pMesh_, Unsigned const p2loop_ = 0 ) ;

  void set_loop(P2M & Mesh_, Unsigned const p2loop_ = 0) ;
  void set_loop(P2M * Mesh_, Unsigned const p2loop_ = 0) ;

  void begin(void) ;
  bool end_of_loop(void) ;

  Iterator<P2OBJ> const & operator ++ (void) ;

  // smart pointer stuff
  P2OBJ const * operator -> () const ;
  P2OBJ       * operator -> () ;
  P2OBJ const & operator *  () const ;
  P2OBJ       & operator *  () ;

} ;

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

Constructors

```
Iterator(void)
Iterator(P2M & Mesh_, Unsigned const p2loop_ = 0 )
Iterator(P2M * pMesh_, Unsigned const p2loop_ = 0 )
```

```
set_loop void set_loop(P2M & Mesh_, Unsigned const p2loop_ = 0)
         void set_loop(P2M * Mesh_, Unsigned const p2loop_ = 0)

   begin void begin(void)

end_of_loop bool end_of_loop(void)

      ++ Iterator<P2OBJ> const & operator ++ (void)

      -> P2OBJ const * operator -> () const
         P2OBJ       * operator -> ()

       * P2OBJ const & operator * () const
         P2OBJ       & operator * ()
```

4 The internal ordering of the mesh data set

4.1 Boundary and internal instances of basic entities

A PDE problem usually requires the specification of a suitable set of boundary conditions. This issue, which is necessary to ensure the correct mathematical statement of the problem, may strongly affect practical implementations. Actually, in numerical algorithms, some special treatment is normally deserved to data referring to the boundaries of the computational domain. Hence, in any numerical application code a distinction is made between “internal” and “boundary” instances of basic entities. The former ones are located inside the computational domain, while the latter ones are either located on the geometric boundaries (vertices and edges) or adjacent to them (as in the case of “boundary” polygons). The topology of a geometrical object is strongly affected by this feature. For example, a boundary vertex belongs to at least one boundary edge; a boundary edge belongs to a boundary polygon, while an internal edge is always shared by two internal polygons; an internal polygon can not have any boundary edge in its instantiation, but it may have boundary vertices, and so on. A naive solution, which is usually suggested in C++ text books, would consist in designing specific classes for boundary and internal objects, by inheriting from an abstract class or a class in a higher position in the class hierarchy. A more effective solution might

equip the mesh representation with the capability of returning the information whether a given instance is internal or located on the boundary. This choice leaves the user the possibility of organizing its program in accord with the real needs of the application. For example, if many loops on boundary entities must be performed, the program can build a list of pointers to those entities and loops on the items of this list.

4.2 Renumbering geometric objects

There are several different ways of implementing a separate access to internal and boundary entities, for example by enumerating in different list structures the entity identifiers. However, a very simple and efficient strategy consists in using an internal representation of the mesh data structures which is strongly ordered, in such a way that each instance of an elementary entity has a very precise location in memory, and can be easily and very fast accessed in random way.

Vertex-, edge-, and polygon-type instances in a mesh data set are numbered consecutively starting by zero[5]. Each entity is thus mapped onto a subset of the natural numbers, and this mapping will be called *the rank*, and identified in the sequel by the symbol $\mathcal{R}$. The mesh data set is equipped with the corresponding functionality which returns the rank of any given entity. Analogously, vertices in edge and polygon instances and edges in polygon ones are locally numbered consecutively starting from zero, and these data structures are also equipped with functionalities to return this kind of information.

Since a direct correspondence may be established between the mesh rank of an entity and its position within a list or an array structures – in a FORTRAN implementation it should be nothing but integer indices of arrays – the boundary instances may be forced to be stored by a suitable renumbering before – or eventually after – the internal ones.

For example, if v_1 is a boundary vertex and v_2 is an internal vertex, their ranks will satisfy the relation

$$\mathcal{R}(\mathsf{v}_1) < \mathcal{R}(\mathsf{v}_2)$$

By using similar constraints, the boundary triangles in the example of figure 2 on the left may be reordered as shown on the right, where we have considered only the subgraph formed by boundary vertices and edges.

[5]This is the standard default in C and C++ programming language.

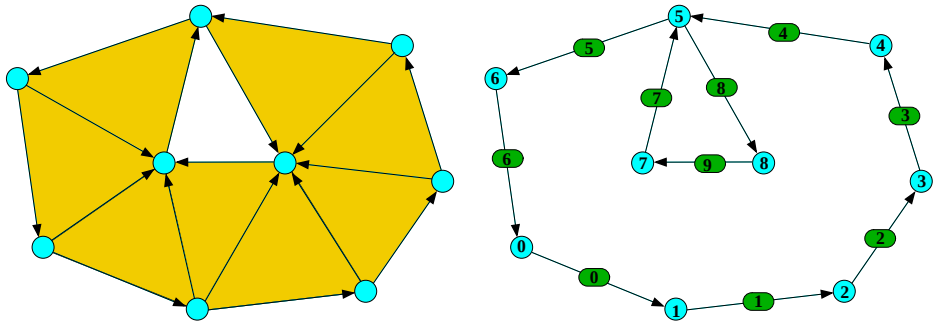

Figure 2: Mesh and boundary graph

The definition of a well-ordered mesh

In its actual implementation, P2MESH uses the internal representation of the basic mesh data structures, which fulfills the constraints previously reported. A mesh which satisfies all these constraints is conventionally called *a well-ordered mesh*. The P2MESH software library automatically builds an internal representation of this kind when the application using the library starts the execution.

We here indicate the basic operations which are performed:

(a) Vertex numbering: mesh boundary vertices are reordered in such a way that when $\mathsf{e} = (\mathsf{v}^a, \mathsf{v}^b)$ is a boundary edge, then

$$\mathcal{R}(\mathsf{e}) \geq \mathcal{R}(\mathsf{v}^a)$$

(b) Edge numbering: mesh boundary edges are reordered in such a way that when for the two consecutive edges

$$\mathsf{e}^a = (\mathsf{v}^a, \mathsf{v}) \qquad \text{and} \qquad \mathsf{e}^b = (\mathsf{v}, \mathsf{v}^b)$$

the vertex v is not contained in any other boundary edge, one and only one of the two next conditions is satisfied

1. $\mathcal{R}(\mathsf{e}^a) + 1 = \mathcal{R}(\mathsf{e}^b)$;
2. $\mathcal{R}(\mathsf{e}^a) > \mathcal{R}(\mathsf{e}^b)$.

(c) Boundary edge orientation: let $\mathsf{e} = (\mathsf{v}^0, \mathsf{v}^1)$ be a boundary edge, then the external region is on the right.

(d) Polygon numbering: mesh boundary polygons are reordered is such a way that when p^0 and p^1 are boundary polygons and $\mathsf{e}^0 \in \mathsf{p}^0$, $\mathsf{e}^1 \in \mathsf{p}^1$ are boundary edges, then

$$\mathcal{R}(\mathsf{e}^0) < \mathcal{R}(\mathsf{e}^1) \qquad \Longrightarrow \qquad \mathcal{R}(\mathsf{p}^0) \leq \mathcal{R}(\mathsf{p}^1)$$

notice that we use "$\leq$" instead of "$<$" because p^0 and p^1 can be the same polygon.

The reordering algorithm reads as

Algorithm *Reordering Mesh Algorithm*
Input: a mesh, given by the ordered set vertices, $\mathcal{V} = \{v_i\}$, edges, $\mathcal{E} = \{e_i\}$, and polygons, $\mathcal{P} = \{p_i\}$
Output: the ordered set of boundary vertices, $\partial\mathcal{V}$, of boundary edges, $\partial\mathcal{E}$, and boundary polygons, $\partial\mathcal{P}$

```
1.  (* builds the set of boundary edges M in any order, and reverse them if necessary *)
2.  M ← ∅
3.  foreach e ∈ E
4.          do if e.p_0 = nil then
5.              (* reverses the edge orientation *)
6.              if e.p_1 = nil then
7.              M ← M ∪ {e}
8.  (* renumbers boundary edges, vertices and polygons *)
9.  foreach e ∈ M
10.         do if e non visited
11.             then (* Save the first vertex pointer *)
12.                 v_a ← e.v_0
13.                 repeat
14.                     mark e as visited
15.                     ∂V ← ∂V ∪ {e.v_0}
16.                     ∂E ← ∂E ∪ {e}
17.                     if e.p_0 ∉ ∂P then ∂P ← ∂P ∪ {e.p_0}
18.                     (* find e' ∈ M such that e.v_1 = e'.v_0 *)
19.                     e ← e'
```

20. **until** $v_a \neq e, v_0$
21. **return** $\partial\mathcal{V}$, $\partial\mathcal{E}$, $\partial\mathcal{P}$.

The algorithm searches for the boundary entities and renumbers them in accord with the ordering relationships previously exposed. The computational complexity is everywhere linear, with the exception of the search algorithm in line 18, which is quadratic. However, this limitation is not so stringent, because the number of boundary element upon which the search is performed is of the order of the square root of the total number of elements, thus the resulting reordering algorithm has still linear complexity with respect to the total number of mesh elements.

References

[1] BARTON, J. J., AND NACKMAN, L. R. *Scientific and engineering C++*. Addison-Wesley Pub. Comp., 1994.

[2] FURNISH, G. Disambiguate glommable expression templates. *Computer in Physics 11(3)* (1997), 263–269.

[3] FURNISH, G. Container-free numerical algorithms in c++. *Computer in Physics 12(3)* (1998), 258–265.

[4] VELDHUIZEN, T. L. The Blitz++ Home Page, 1999. http://oonumerics.org/blitz/.