ISSN: 1121-0540

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

PAVIA
1998

PUBBLICAZIONI

N. 1124

Enrico Bertolazzi and Gianmarco Manzini

Template classes for PDE solvers on 2-D unstructured meshes.

Template classes for PDE solvers on 2-D unstructured meshes.

Enrico Bertolazzi and Gianmarco Manzini

The software library P2MESH is a collection of C++ class templates suitable for developing prototypes of high-performance PDE solvers on unstructured 2-D meshes. P2MESH together with its underlying data structures are designed to support a wide variety of discretization methods on triangles and quadrilaterals, such as Finite Volumes or Finite Elements. Both stationary and time-dependent problems can be addressed. The design philosophy of P2MESH does not consider neither specific model problems nor built-in approximation algorithms. The software package turns out to be of general purpose and it may also be used as a building block in the implementation of numerical codes both for engineering applications and mathematical problems.

Additional Key Words and Phrases: Object-Oriented programming, Finite Elements, Finite Volumes, PDE solvers, unstructured mesh.

1. INTRODUCTION

Unstructured grids in PDE calculations allow some families of numerical methods – such as Finite Elements (FE) and Finite Volumes (FV) – to be applicable in problems involving general geometries. As pointed out by Berzins et al. [1998], this issue is nowadays one of the most important trends in the development of numerical software packages for PDEs. For example, we can mention a (largely incomplete) list of remarkable software products such as DIFFPACK [Bruaset and Langtangen 1996], KASKADE [Beck et al. 1995], UG [Bastian et al. 1997], and SPRINT2D, SPRINT3D [Berzins et al. 1998]. This broad applicability is counterbalanced by the necessity of managing *unstructured triangulations* in an effective and efficient way. The computational domain is partitioned in a set of basic geometrical entities such as triangles and quadrilaterals in 2-D, or tetrahedrons and prisms, in 3-D. Managing such entities is trivial in some simple situations (for instance when dealing with a mesh of regular rectangles covering a square domain), but it may be really difficult when an unstructured mesh is considered on a domain of general shape, because topological and geometrical information must be correctly represented and coherently manipulated by the program.

Surprisingly enough, few attention has been apparently given by numerical mathematicians to this issue, more often considered a minor implementation detail and felt as the task of developers of mesh generation algorithms. In fact, there is a considerable literature about methods for automatic grid generation both in 2-D and 3-D, see for instance [George

Name: Enrico Bertolazzi
Affiliation: Department of Mechanics and Structures Engineering, University of Trento
Address: via Mesiano 77, I – 38050 Trento, Italy

Name: Gianmarco Manzini
Affiliation: institute of Numerical Analysis – CNR
Address: via Abbiategrasso 209, I – 27100 Pavia, Italy

1991; George 1996]. There is also an amount of scientific production pertaining to object-oriented (OO) abstractions for FE methods, especially for structure analysis, see [Zimmermann et al. 1992; Zeglinsky and Han 1994; McKenna 1997; Dubois-Pèlerin and Pegon 1997; Dubois-Pèlerin and Pegon 1998; Eyheramendy and Zimmermann 1998].

Instead, to our knowledge there seems to be very few published material regarding the kind of abstractions that, at the mesh definition level, are conceptually useful in developing and implementing computationally efficient FE and FV algorithms on unstructured meshes. In this work we claim this point is indeed crucial. The effectiveness and flexibility of a given implementation are strongly dependent on how efficiently the data sets based on the mesh adopted abstraction are accessible during the computation. Our extensive bibliography research has discovered only one paper concerning mesh abstractions for FE or FV algorithms, i.e. Simpson [1997]. This author studies an abstract relational data model for planar `2-D` triangular meshes, which is conceptually half-way between the geometric abstractions of a mathematical description and the list representation of a source code implementation. Our work has been partially inspired by the above idea, although it is more oriented towards the real implementation of a numerical algorithm. Basically, all relevant information to be represented and manipulated by a numerical solver, such as geometrical quantities, physical unknowns and auxiliary dependent variables, are logically associated to an elementary mesh *entity*, namely a *vertex*, an *edge*, or a *polygon* (*triangular* or *quadrilateral*). Therefore, mesh-based application-dependent types are built by using an algorithm-oriented mesh abstraction as follows. A generic mesh-based type, containing all geometrical information, is parameterized by a set of application-dependent types. The mesh itself is an extension of a generic type, which encapsulates a set of containers for objects of mesh-based application-dependent types.

An example clarifies the working mechanism. One considers a generic geometric type such as a *mesh vertex*, and extends it into a more complex data type, i.e. a FE *node* in such a way the properties of the basic type also hold in the extended type. For instance, we expect that a `2-D` mesh vertex has two real numbers as coordinates, and a set of methods to manipulate it, that is to retrieve its coordinate values, to find connectivity, if any, with other vertices, and so forth. Vertex attributes and methods are still present in the extended FE node to manage transparently the underlying mesh. Furthermore, the extended FE node contains also the data and the functionalities required by the problem and by the approximation algorithm under consideration.

The software library `P2MESH` actually embodies the above concept. The library is a collection of `C++` class templates which are suitable for *generic* data abstraction and object-oriented programming. It has been designed to support the implementation of complex mesh-based data types for a wide variety of unstructured FV and FE discretization methods on unstructured `2-D` meshes. The name "`P2MESH`" is not an acronym, but stands for "mesh of polygons in `2-D`" because it processes either triangular or quadrilateral meshes. `P2MESH` is a general purpose package, and it does not make reference either to any specific family of PDE problems or to any particular class of approximation techniques.

One of the most important features we took care of regards the non-procedural nature of this programming model. In this respect, OO programming techniques turn out to be quite effective: in fact, they are versatile in the description of data abstractions and helpful in isolating the data structure design from the implementation.

The big challenge for a numerical object-oriented library still remains the computational efficiency. A numerical procedure usually repeats many thousands (or millions) of times the same sequence of actions, identically applied to the most elementary entities of the discretization involved, such as, to name a few, control volumes, edges and points.

For this reason, much of the current programming practice in scientific computing is *ad hoc* and basically procedural. In a procedural programming model, data structures are directly accessed and elaborated by external procedures that treat them as input/output arguments. This approach allows several straightforward code optimization techniques: vectorization of loops, loop unrolling, etc. By contrast, in a non-procedural OO programming model *different functionalities which operate upon different data are themselves part of the data specification* [Stroustrup 1998]. Aggregates of data and functions performing operations on data – also known as *abstract data types* [Coplien 1992] – enable the software organization according to a high level of abstraction. The main advantage is that reusable and maintainable software is easily developed. Unfortunately, traditional optimization techniques for procedural programs are disappointingly inefficient whenever applied to nested aggregates of data and temporary objects, which are common features in an object oriented language, such as C++ [Veldhuizen 1995b]. A special programming technique, known as *static polymorphism*, has been used to prevent performance penalties, see Furnish [1997, 1998] and Veldhuizen [1995b, 1995a].

The present article illustrates the non-procedural programming paradigm supported by P2MESH and also aims at giving the reader a fairly high-level acquaintance with the library capabilities.

The outline of the paper is as follows. In section 2 we present the mathematical background and discuss some common features of FE and FV methods. In section 3 we state the mesh abstraction used in the object-oriented design of P2MESH and discuss the graph interpretation. In section 4 we give an overview of the library implementation and in section 5 we discuss some technical details. In section 6 we analyze the example reported in appendix, which illustrates how P2MESH applies in developing a simple Poisson solver program. Finally, in section 7 we state the conclusion.

2. BASIC ABSTRACTIONS IN FE AND FV PROGRAMMING

In this section we briefly present the mathematical framework P2MESH refers to. Some issues concerning data abstractions in FE and FV methods for PDEs are sketched and the general model of a PDE solver, whose implementation is the engine of P2MESH, is then introduced. It is important to note that the differential problems and the approximation techniques discussed throughout this section are possible applications, but do not play any role in the library implementation.

Both FE and FV approximations are based on a mesh partitioning $\mathcal{T}_h$ of the computational domain. The geometric elements $K \in \mathcal{T}_h$ are part of the formal definition of a *finite element*, see [Ciarlet 1987]. They are also called “control volumes”, “finite volumes”, or simply “cells”.

A great amount of literature, either papers or textbooks, covers both theoretical and practical aspects of FE and FV methods. An extensive presentation of the matter is beyond the scope of the present paper. The interested reader is referred to the general and didactic presentations of FE in [Ciarlet 1987] and of FV in [Hirsch 1990].

2.1 Basic abstractions in Finite Element Methods

Every FE method relies on a discrete variational formulation of a differential problem, consisting basically in the definition of an appropriate functional space V_h, whose properties must be suitable both for theoretical analysis and for practical calculations. The space V_h is usually composed by functions whose restriction to any element $K \in \mathcal{T}_h$ is a polynomial of a prescribed order. It is crucial that in V_h there exists a set of basis functions $\{v_i\}$ having a small support, which greatly simplifies the computation of the integrals involved in the variational formulation. The approximate solution u_h is the linear combination of the basis functions $\{v_i\}$, that is $u_h = \sum_{i=1}^{N} u_i v_i$. The N entries in $\mathbf{u} = \{u_i\}$ are the unknowns to be determined and are generally called the *degrees of freedom* of the method. They are the solution of a linear system of the form $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is built by using the stiffness matrix and the – eventually lumped – mass matrix and $\mathbf{b}$ the load vector (this terminology comes from the theory of elasticity).

The meaning of the *degrees-of-freedom* is affected by the choice of the basis functions $\{v_i\}$. For example, in Lagrangian-type FEs the degrees-of-freedom approximate the values of the solution at a given set of nodes within K; in Hermitian-type FEs, the degrees-of-freedom approximate also the values of the directional derivatives at some given locations in K.

A common strategy in implementing FE methods consists in assembling $\mathbf{A}$ and $\mathbf{b}$ from local contributions estimated on every mesh element, [Johnson 1990], namely:

$$A_{i,j} = \sum_{K \in \mathcal{T}_h} A_{i,j}^{(K)}, \qquad b_i = \sum_{K \in \mathcal{T}_h} b_i^{(K)}.$$

Essential boundary conditions, dependent on the physical problem and on the variational formulation, are usually considered at this stage. Finally, a library solver for linear problems is called to compute the degrees-of-freedom $\{u_i\}$ and then the approximate solution u_h can be post-processed.

In a typical procedural implementation, a *loop* is performed over all the elements of the triangulation. In an OO approach, it seems natural to utilize *element-* and *node-based* data structures. A suitable set of iterator types provides a way to access sequentially the attributes of the current element object and the connected node objects via the functions of the public interface.

The scientific production about OO programming for FE methods has largely concentrated on identifying the main class abstractions to be used in FE models in structure analysis. The abstractions outlined in literature basically consist of the following classes: Node, Element, Constraint, Load, and Domain, see [Zimmermann et al. 1992; Dubois-Pèlerin and Pegon 1997; McKenna 1997].

These classes are used to describe a FE model and to store the results of a FE analysis. They can be combined with other classes to form the governing equations and to handle the numerical operations in the solution procedures, see [Zeglinsky and Han 1994].

We emphasize that all of the classes listed above convey a precise geometrical meaning. This is evident for objects of type Node and Element, which correspond to vertices and elements of a mesh. The types Constraint and Load may be associated to the types Node and Element, [Dubois-Pèlerin and Pegon 1998]. The type Domain, as pointed out in [Dubois-Pèlerin and Pegon 1997], may be associated to the mesh triangulation of the computational domain. In all of the previous cases, associations may imply either *is-a* or *has-a* relationships.

2.2 Basic abstractions in Finite Volume Methods

Opposite to the case of Finite Elements, the set of control volumes $\{K\}$ is not necessarily identified with $\mathcal{T}_h$. Indeed, the FV methods on unstructured grids proposed in literature actually deal with two different kinds of meshes: the *primal mesh*, that is $\mathcal{T}_h$, and its *dual mesh*.

For the sake of exposition and with no pretence at completeness, the FV methods will be regrouped into the two following families, depending on which mesh the control volumes are considered in:

—a *cell center* FV method takes the control volumes as the elements of the primal mesh;

—a *cell vertex* FV method takes the control volumes as the elements of the dual mesh.

The relation between $\mathcal{T}_h$ and its dual is based on the association between the vertices, the edges and the centroids of the two meshes. This association is not unique, i.e. more than one dual mesh can be specified from a given primal mesh. As a result, many variants of the cell vertex FV methods exist and are documented in literature.

The FV discrete formulation can be formally obtained by integrating the divergence form of a PDE over the generic control volume K, and then applying the divergence theorem. The cell-average value of the unknown $\mathbf{U}$, i.e. $\overline{\mathbf{U}}_K = \frac{1}{|K|}\int_K \mathbf{U}$, is determined by the distribution of the flux density $\mathbf{n} \cdot \mathbf{F}$ on the cell interface boundary ∂K.

Flux integral computations are performed by a numerical flux model, which takes into account the contributions from both sides of any internal edge and the boundary conditions on boundary edges.

The integral formulation of the original differential problem is the same when the finite control volume is identified with the cell of the primal mesh or with the cell of the dual mesh. Furtherly, the discrete FV method requires the same kind of operations:

—for every (primal or dual mesh) edge, the estimation of the contribution of the numerical flux to the residual of the control volumes sharing that edge;

—for every (primal or dual mesh) control volume, the updating of the cell-averaged solution.

Nevertheless, the program design and its practical implementation may substantially differ in the two cases. In a cell center FV method, the control volumes are the primal mesh cells. Therefore, the solver implementation generally utilizes a first loop on the edges and a second loop on the cells of the primal mesh. In a cell vertex FV method, the control volumes are the dual cells associated to the primal mesh vertices. Hence, the solver implementation generally utilizes a first loop on the primal mesh cells closed to the primal mesh vertices, and a second loop on the primal mesh vertices.

In the former situation, the program must be capable of retrieving efficiently the information stored in objects of type edge and cell; in the latter one, in objects of type cell and vertex.

The duality in the discrete formulations reflects also in the OO programming models of FV methods proposed in literature, and in the set of class abstractions there outlined. In [Gloth et al. 1998], the authors distinguish between *pointwise* and *edgewise* operations, where a *point* is the center of the control volume in the dual mesh. This software package, which implements a family of FV methods, is based on an hierarchy of classes that encapsulates this kind of operations.

2.3 A mesh-based generic kernel

The analysis carried out so far outlines the fact that the basic FE and FV data types show a geometrical meaning and suggest they are definable as the parametric extensions of the corresponding geometric types. Access operations consist in retrieving topological information such as connectivity and geometrical information such as the coordinates of the vertices, the length of the edges, or the area of the element and its neighbors. The present abstraction holds true for both FE and FV methods. It is worth noting that formal analogies between FE and FV formulations have been remarked earlier in literature [Selmin and Formaggia 1996].

The identification and the actual implementation of these basic types and of their related access operations is the kernel of a programming framework truly independent of how a numerical algorithm works. A generic library can thus be envisaged, amenable of extensions to a broad range of mathematical and numerical applications [Musser and Stepanov 1994; Bader and Berti 1998]. From a technical viewpoint, genericity is expressed by static polymorphism via the C++ mechanism of instantiating template classes from class templates [Barton and Nackman 1994]. The recursive template pattern ensures efficiency in algorithm-oriented software design [Furnish 1997; Furnish 1998; Veldhuizen 1999]. *Iterator types* are provided to make possible an easy and efficient access to the local data structures, as in the spirit of the Standard Template Library (STL) design [Musser and Saini 1996]. These STL-originated iterators are also wrapped in a set of templated iterator types whose usage syntax should be more suitable to PDE applications.

The generic solution process can be summarized by the following three steps:

—a *global discrete version* of a differential operator is first given on the whole computational domain; loosely speaking, this step coincides with the definition of the numerical scheme to be implemented;

—a *local representation* of the discrete operator is defined in such a way that its global form is built from local contributions;

—the *approximation to the solution* is estimated via either local update operations (whenever possible) or by assembling and formally inverting the global discrete operator.

Boundary conditions may be taken into account in all of the former steps. The second step is usually designed by means of loop cycles on specific sets of geometric entities and may also require some data manipulations, such as data movements, exchanges or comparisons. Finally, the third step requires a global assembling, which relies on some mapping from a local to a global numbering system of geometric entities. Some external modules such as linear algebra solvers may also enter the solver design.

3. THE MESH REPRESENTATION

The central issue in numerical approximation to PDE problems by either FE or FV methods is the triangulation of the computational domain. Triangulations may be of *structured* and *unstructured* type, the difference essentially being in the way information about mesh *connectivity* and *topology* are given and available for processing. Following the presentation in [George 1991; George 1996], we define *topology* as the description of an element (including the description of its edges) in terms of its vertices. The topology of a mesh is the topology of all its elements. The mesh connectivity is the description of the way vertices, edges, and polygons are connected, that is which is neighbor to which. In an unstructured mesh this kind of information is explicitly given by special *connectivity lists*. On the contrary, the connectivity of a structured mesh is implied from the knowledge of integer identifiers of mesh entities. `P2MESH` actually deals with unstructured triangulations $\mathcal{T}_h$ supposed *conformal* in the sense of [Ciarlet 1987], i.e. the set of elements $K \in \mathcal{T}_h$ covers the domain Ω in "continuous" sense so as distinct elements do not overlap or intersect.

It is clear there is no unique way of representing a mesh. The multitude of representations is a consequence of the wide range of problems we want to take into account. In addition, for the same application there may be more than one solution approach. These various approaches may lead to implementations which differ significatively in the way mesh related information is used. Of course, a general purpose tool that assumes no special features will cope with all possible applications but not as efficiently as one that exploits special features that an application may possess. Regardless of these arguments, we claim that, as pointed out by [George 1996], the minimum data set to represent a mesh should consist in

(i) the list of vertex coordinates and

(ii) the connectivity table "elements $\rightarrow$ constitutive vertices".

All of the topological relationships among mesh geometrical entities, such as vertices, edges, and polygons, can be reconstructed from data set (ii) and the values of geometrical quantities such as distances between vertices, components of normal vectors, edge lengths and element areas can be computed using (ii) and the values of the vertex coordinates in (i). In a typical procedural implementation, this information is available as an ordered collection of real values and tables of integer identifiers, but our non-procedural approach needs a more sophisticated mesh abstraction. In the remaining of the section, we devise a formal definition of an unstructured `2-D` mesh and we discuss some design aspects.

3.1 A formal definition of an unstructured `2-D` mesh

The mesh abstraction used in `P2MESH` is based on a graph interpretation and the notations and conventions used in this section are very similar to the ones commonly adopted in the context of graph theory. Also for the definitions of several basic keywords, such as "planar graph", "oriented edge", "arc", "cycle" and others we refer the reader to graph theory text books; see for example [Christofides 1975]. It is to be noticed that the notion of polygon given in the sequel is based on the concept of "bounded region of a planar graph", which is a subgraph whose vertices form a cycle.

An unstructured `2-D` mesh can be interpreted as a planar graph $\mathcal{G}(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Vertices are generally considered as primitive entities, and no definition is usually provided for them. Edges are defined as an irreflexive binary relation on $\mathcal{V}$. In what follows, the formal definitions of **vertex**, **edge**, **polygon** and **mesh** are derived from the graph notion earlier introduced.

DEFINITION 1. *A **vertex*** $\mathtt{v} \in \mathcal{V}$ *is an ordered pair of real numbers* (x, y), *which are called* the coordinates.

DEFINITION 2. *An **edge** is a connection between two vertices, and is defined by the oriented pair* $\mathtt{e} = (\mathtt{v}^0, \mathtt{v}^1)$. *In a planar graph, connections are represented as non-intersecting arcs. The definition is usually further restricted by admitting as possible arcs only straight segments.*

DEFINITION 3. *A **polygon** is a bounded region of polygonal shape, defined through a counterclockwise ordered list of mesh* vertices $(\mathtt{v}^0, \mathtt{v}^1, \ldots, \mathtt{v}^{n-1})$*, actually the vertices of the polygon. A polygon must not contain any mesh vertex in its interior. The cardinality of the vertex list is referred to as the **size** of the polygon.*

A mesh is basically a set of containers for **vertex**-, **edge**-, and **polygon**-type instances, that is

DEFINITION 4. *An **unstructured mesh** is an oriented graph composed by instances of vertices and polygons (definitions 1 and 3). A mesh can be equivalently seen in the* 2-D *case as a collection of vertices and edges (definitions 1 and 2).*

The formal definition is completed by introducing the access mechanism to specified subsets of the mesh data sets provided by *iterators*. Since the actual implementation of P2MESH is based on the STL *vector* container, iterators are easily available, allowing both sequential and random access.

REMARK 1. *In the definition of a graph not all of the possible connections among couples of vertices belong to the graph. Similarly, in the definition of a mesh not all of the polygons consistent with definition 3 must be necessarily present in the mesh data set. A "legal" polygon might be a "hole" in the computational domain and would not appear in the list of mesh polygons.*

In spite of redundancy and at the price of a greater memory storage load, further data are considered by P2MESH, whose direct availability yields better efficiency in practical implementations. Any instance of *polygon*-type also contains the counter-clockwise ordered list of pointers to its edges $(\mathtt{e}^0, \mathtt{e}^1, \ldots, \mathtt{e}^{n-1})$. Any instance of *vertex*-type may also contain the connectivity lists of pointers to connected vertices, to incident edges and to adjacent polygons. Being such an information not commonly used by many FE and FV algorithms, its availability in P2MESH is not provided by default, but left as an option for the user. Any instance of type *edge* is also capable of returning pointers to adjacent polygons. In order to limit memory occupation, this information is not stored in an independent list, but pointers are calculated afresh when accessed.

3.2 The internal ordering of the mesh data sets

Boundary and internal instances of basic entities. Special mathematical treatment is normally provided to the boundary of the computational domain. For this reason, it is useful to distinguish between an "internal" and a "boundary" instance of a basic geometric type. The definition of a boundary vertex and a boundary edge are obvious, while a "boundary polygon" is a polygon with at least one boundary edge.

The topology of a geometrical entity in the mesh is strongly affected by this feature. For example, a boundary vertex belongs to at least one boundary edge. A boundary edge belongs to at most one boundary polygon; by contrast an internal edge is always shared by two internal polygons. An internal polygon can not have any boundary edge, but it may have boundary vertices.

To manage efficiently this kind of information, a rather classical C++ strategy relies on using inheritance to specialize a class in a higher position of the class hierarchy – usually an abstract class – and then applying virtual functions and dynamic polymorphism [Vankeirsbilck and Nelissen 1994; McKenna 1997].

From the viewpoint of a generic library, we believe that a more effective solution consists in giving the mesh representation the capability of detecting whether an instance is an internal or a boundary item.

This design issue does not force the user any particular implementation choice and enables the possibility of organizing the final code according to the choice of the user.

Re-ordering geometric objects. There are several different ways of implementing separate access to internal and boundary entities, for example by enumerating in different list structures the entity identifiers. P2MESH adopts a simpler strategy which works efficiently in 2-D: a special numbering of all the data structures in the mesh. In such a way, any instance of type vertex, edge or polygon has a very precise location in memory and can be quickly accessed in random way.

Vertex-, edge-, and polygon-type instances in a mesh data set are consecutively stored in memory starting by a zero offset address. 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 public method `local_number` which returns the rank of a given entity. Analogously, vertices in edges and polygons and edges in polygons can be considered as locally numbered in a consecutive way starting from zero. These data structures are also equipped with their version of the public method `local_number`.

P2MESH forces the boundary instances to be stored *before* the internal ones by applying a suitable reordering strategy during mesh construction. The word “before” in the previous phrase is motivated by an implementation choice; at the same place the word “after” would work equally well. In a FORTRAN implementation the rank would be nothing but an integer index of an array and reordering would mean the renumbering of integer identifiers. However, since the implementation of P2MESH is internally based on C++ memory pointers, we prefer the usage of the term “reordering” instead of “renumbering”. The constraints which are satisfied by the new order are:

(a) vertex reordering: 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 reordering: 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, only one of the two following conditions must hold:
1. $\mathcal{R}(\mathsf{e}^a) + 1 = \mathcal{R}(\mathsf{e}^b)$,
2. $\mathcal{R}(\mathsf{e}^a) > \mathcal{R}(\mathsf{e}^b)$;

(c) polygon reordering: boundary polygons are reordered in 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

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

(we use “$\leq$” instead of “$<$” because p^0 and p^1 can be the same polygon).

Furthermore, boundary vertices and edges are ordered in such a way that if $\mathsf{e} = (\mathsf{v}^0, \mathsf{v}^1)$ is a boundary edge the external region is on the right. As a consequence of these constraints, if for example v_1 is a boundary vertex and v_2 is an internal one, their ranks will satisfy the condition $\mathcal{R}(\mathsf{v}_1) < \mathcal{R}(\mathsf{v}_2)$. Similar relations hold true also for the other mesh data sets.

Whenever a mesh is run-time instantiated from input data by P2MESH mesh builders, a reordering algorithm is automatically and transparently invoked. The algorithm is detailed in [Bertolazzi and Manzini 1999a]. Since it is based on the quick-sort and the binary search – available in the STL – its computational cost has a growth proportional to $n \log n$, where n is the total number of boundary items [Knuth 1998; Hoare 1962]

4. OUTLINE OF THE LIBRARY

This section presents an overview of the generic classes defined in P2MESH and of the way they can be used to build complex numerical applications. The main features of the library are presented in some details, but for a complete description – namely, function prototypes and examples – we refer the interested reader to the programmer's manual [Bertolazzi and Manzini 1999b] also available in the package distribution.

In the philosophy of P2MESH, the types of data sets used in a numerical solver are defined by inheritance from the corresponding geometric types provided by the library. Thus, the core of the P2MESH software library comprises five *generic* base classes, which are templates for the definition of specific derived classes.

In this section, the P2MESH base classes are referred to as *library classes*, and the derived classes as *project classes*, being *the project* the application based on P2MESH. The following table indicates the names of the P2MESH library classes, the conventional names adopted for the project classes in the documentation, and the nature (geometrical or not) of the type.

Base Class Name	Derived Class Name	Parameterized Type
p2_common	Common	shared information
p2_vertex	Vertex	vertex instance
p2_edge	Edge	edge instance
p2_poly	Poly (Triangle, Quad)	polygon instance
p2_mesh	Mesh	mesh instance

It should be noted that the project class names are simply a conventional choice: no name is forced to the types defined by the library user. The prefix p2_ in the base class names has been adopted in the library implementation to avoid name conflicts with other project names, defined either by the user or by others software packages.

Notice that from the base class p2_poly at least two different types of polygons may be derived, either triangles or quadrilaterals. The class name 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 library classes are designed to be used as base classes in public derivations; no objects of this types should be instantiated. The instances of the project classes Vertex, Edge, Poly – Triangle, Quad– and Mesh will be simply called *vertices*, *edges*, *polygons* – *triangles*, *quadrilaterals* – and *meshes*. No objects of type Common must be instantiated.

The keyword const is omitted in the prototype declarations reported herein for the sake of compactness. However, the complete form of the declarations can be easily found in the kernel description given in [Bertolazzi and Manzini 1999a], or in the source code.

4.1 The library class p2_common

The class p2_common is the base class for the public derivation of the class Common. The class Common contains the static data and the typedef aliases shared by the project classes. It also inherits the statically stored information specified in the template argument list of p2_common. The template header declaration of the class p2_common is:

```
template <
  typename P2V_type,                  // vertex   project name
  typename P2E_type,                  // edge     project name
  typename P2P_type,                  // polygon  project name
  typename P2M_types,                 // mesh     project name
  unsigned SIZE_value    = 3,         // polygon  size
  bool     LIST_value    = false,     // vertex   connectivity
  typename REAL_type     = double,    // floating point type
  typename INTEGER_type  = int,       // integer  type
  typename UNSIGNED_type = unsigned,  // unsigned integer type
  typename VMARK_type    = unsigned,  // vertex   marker type
  typename EMARK_type    = unsigned,  // edge     marker type
  typename PMARK_type    = unsigned>  // polygon  marker type
class p2_common ;
```

The default values assume a triangular mesh, no vertex topology lists, standard built-in arithmetic types and unsigned integer markers:

```
class Common : public p2_common<Vertex, Edge, Triangle, Mesh> { } ;
```

A quadrilateral-based mesh requires that the entry SIZE_value be explicitly set to 4 in the template argument list of the base class p2_common:

```
class Common : public p2_common<Vertex, Edge, Quad, Mesh, 4> { } ;
```

Whenever set to true the entry LIST_value makes available the vertex connectivity in the mesh representation. The double mechanism of template parameterization of the base classes with the project class Common and inheritance from them allows the redefinition of the arithmetic types in all of the classes built on P2MESH, by changing simply the entries REAL_type, INTEGER_type and UNSIGNED_type in the template argument list of p2_common. The arithmetic types in the library classes are parameterized in the template argument list of p2_common and are visible as Integer, Unsigned and Real, with default values the built-in types int, unsigned and double.

It is a common practice in programming numerical algorithms to assign a marker or a set of markers to vertices, edges and polygons. A marker is generally used to drive a specific process, for example the treatment of the boundary conditions. P2MESH provides the final user with several mesh builder methods, which initializes the mesh data set. A marker is by default an unsigned integer, but it may be an instance of whichever type defined by the library user. The name of the marker type may be introduced in the template argument list of p2_common. In this case the operators << and >> must be overloaded for I/O operations. Every mesh builder can take pointers to project-defined functions for processing markers.
p2_common does not have any public methods.

4.2 The library classes p2_vertex, p2_edge, and p2_poly.

The project types Vertex, Edge, and Poly for extended vertices, edges, and polygons are publicly derived from the library classes p2_vertex, p2_edge, and p2_poly by using the syntax

```
class Vertex : public p2_vertex<Common> { ... } ;
class Edge   : public p2_edge  <Common> { ... } ;
class Poly   : public p2_poly  <Common> { ... } ;
```

The project class Common parameterizes the template argument list of all the library classes. The double mechanism of public inheritance and template parameterization allows the library generic classes to be specialized by the project classes [Furnish 1997; Furnish 1998; Veldhuizen 1999].

At the same time, the public interface of p2_vertex, p2_edge, and p2_poly is inherited by the corresponding project classes. The public interface is designed by following some regular and systematic rules with very few exceptions. It provides a small set of simple member functions which can combine in more complicated expressions. The public methods are logically grouped and listed in three subsets:

—*topological* methods : they return information about the first neighbor connectivity of a vertex, an edge, or a polygon to instances of type Vertex, Edge, and Poly;

—*geometrical* methods : they return “non-topological” information, such as for example the coordinates of a vertex, the length of an edge, the area of a polygon, and so forth;

—*mapping* methods: they return the coordinates and the Jacobian matrices for the coordinate transformation from and onto the reference polygon; P2MESH implements the simplest linear coordinate mappings for triangles and quadrilaterals and the methods are available only in the public interface of the class p2_poly.

Topological Methods. The public interfaces of the library classes contain almost the same topological methods, whose list is as follows[1].

1. `unsigned n_vertex()` : number of vertices;
2. `unsigned n_edge ()` : number of edges;
3. `unsigned n_poly ()` : number of polygons;
4. `Vertex & vertex(unsigned i)` : reference to the `i`-th vertex;
5. `Edge & edge (unsigned i)` : reference to the `i`-th edge;
6. `Poly & poly (unsigned i)` : reference to the `i`-th polygon;
7. `unsigned local_number(Vertex & v)` : local identifier of vertex `v` ;
8. `unsigned local_number(Edge & e)` : local identifier of edge `e`;
9. `unsigned local_number(Poly & p)` : local identifier of polygon `p`;
10. `bool ok_poly (unsigned i)` : check whether the `i`-th polygon exists;
11. `bool ok_oriented(unsigned i)` : check the edge orientation.

For each instance of type vertex, edge, and polygon the methods of the public interface give the following information:

[1]For the sake of presentation, the enumeration of the public methods adopted here differs from the one used in the programmer’s manual.

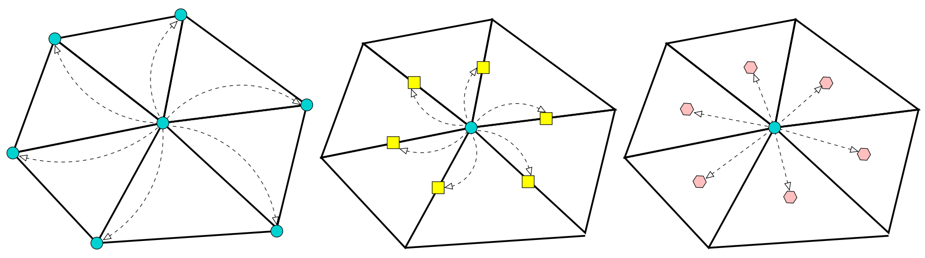

Fig. 1. Vertex–Vertex, Vertex–Edge and Vertex-Polygon connections in the definition of a `p2_vertex` instance (pointers are optionally stored in memory)

—methods 1–3 : *how many* vertices, edges, and polygons are in the instance definition;
—methods 4–6 : their *reference* in the mesh representation;
—methods 7–9 : their *local identifier* in the instance definition.

Figure 1 depicts the pointers to vertices, edges and polygons adjacent to a given vertex. Figure 2 illustrates on the left the local numbering of edges adjacent to a given edge in the mesh, and on the right the local numbering of vertices and edges forming a given polygon and its adjacent polygons. In both pictures a triangle-based mesh is assumed.

In order to save memory storage, the topological methods of a vertex, i.e. methods 1–9 are not available by default, but must be required by introducing `LIST=true` in the template argument list of the class `p2_common`.

Methods 1, 4, 7 in the public interface of the class `p2_edge` return information about the vertices in the current instance of that class. The other methods in `p2_edge` refer to adjacent edges and polygons.

Similarly, methods 1, 4, 7 and 2, 5, 8 in the class `p2_poly` are respectively related to the vertices and edges in the current polygon, while the other ones refer to the adjacent polygon.

Some of the member functions in the public interface return the same information; for example there holds:

```
Poly::n_vertex() == Poly::n_edge() == Poly::n_poly()
```

In spite of redundancy, `P2MESH` provides all of the mentioned methods in order to have the same public interface for all the library classes.

Method 10 checks whether a polygon adjacent a given edge exists. The method is in the public interface of both `p2_edge` and `p2_poly`.

Finally, method 11, which is only in the public interface of the class `p2_poly`, checks the orientation of an edge in a polygon by comparing it with the orientation of an edge as an independent instance of class `Edge`.

Geometrical Methods. We list here the geometrical methods that return non-topological information on vertices, edges, and polygons. These methods are inherited from the classes `p2_vertex`, `p2_edge`, and `p2_poly`.

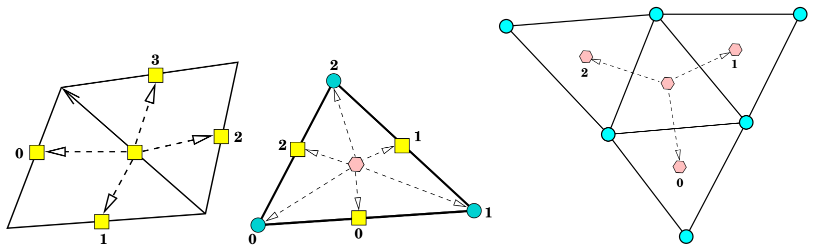

Fig. 2. On the left, local numbering for edges adjacent a given edge in triangles. On the right Polygon-Edge and Polygon-Polygon connections for a triangular `p2_poly` instance.

Methods of vertices

12. `double x()` : first coordinate of the vertex;
13. `double y()` : second coordinate of the vertex.

Methods of edges

14. `double xm()` : first coordinate of the midpoint
15. `double ym()` : second coordinate of the midpoint;
16. `double xt(double & t)` : linearly interpolated first coordinate;
17. `double yt(double & t)` : linearly interpolated second coordinate;
18. `double nx()` : first component of the vector normal to the edge;
19. `double ny()` : second component of the vector normal to the edge;
20. `double length()` : length of the edge.

Methods of polygons

21. `double xc()` : first coordinate of the centroid;
22. `double yc()` : second coordinate of the centroid;
23. `double area()` : polygon area.

The sequence `p2_vertex`, `p2_edge`, and `p2_poly` lists the library classes in increasing order of complexity. In fact, a vertex is the simplest entity existing in a mesh, an edge may be defined by "a couple of vertices" and a polygon by "a set of vertices and edges". It is clearly to be expected that the quantities related to a vertex, e.g. its coordinates, must be available also for the edges and the polygon that vertex belongs to. Thus, the methods returning this kind of information should appear not only in the public interface of the class `p2_vertex`, but also in the classes `p2_edge` and `p2_poly`. However, since an edge contains two vertices, and a polygon three or four, a sorting rule among the vertices must be conceived. For this reason, the prototype of any vertex method in the public interface of `p2_edge` and `p2_poly` is modified by adding an unsigned integer entry at the beginning of its argument list. This entry labels vertices ranging through `0...n_vertex()-1` according to the local numbering. For example, the vertex method `Vertex::x()` becomes `Edge::x(unsigned i)`, or `Poly::x(unsigned i)`.

In a similar way, the methods of edges are in the public interface of the class `p2_poly` and take an additional unsigned entry at the beginning of their argument lists.

Mapping methods. In the class `p2_poly` there is also a minimal set of methods which map the actual polygon – no matter if it is a triangle or a quadrilateral – onto a reference polygon and viceversa. Their prototypes are

24. `st_to_xy(double & s, double & t double & x, double & y)`
    affine mapping from reference to actual element;
25. `void xy_to_st(double & x, double & y, double & s, double & t)`
    affine mapping from actual to reference element;
26. `void jacobian(double & s, double & t, double J[2][2])`
    Jacobian of the mapping;
27. `void inverse_jacobian(double & s, double & t, double InvJ[2][2])`
    inverse Jacobian of the mapping.

4.3 The library class `p2_mesh`

`p2_mesh` is the library class for the public derivation of the project class `Mesh`. As is the case of the other data structures herein examined, the double mechanism of inheritance and template parameterization by `p2_mesh` applies in the definition of `Mesh`.

```
class Mesh : public p2_mesh  <Common> { ... } ;
```

`Mesh` is thus the most complex data structure that can be built by using the library and without any further project specification. Within its private attributes there are the lists of vertices, edges, and polygons, inherited from `p2_mesh`.

4.4 Topological Methods

The main topological information are accessible by the following methods:

1. `unsigned n_vertex ()` : number of vertices;
2. `unsigned n_bvertex()` : number of boundary vertices;
3. `unsigned n_ivertex()` : number of internal vertices;
4. `unsigned n_edge ()` : number of edges;
5. `unsigned n_bedge()` : number of boundary edges;
6. `unsigned n_iedge()` : number of internal edges;
7. `unsigned n_poly ()` : number of polygons;
8. `unsigned n_bpoly()` : number of boundary polygons;
9. `unsigned n_ipoly()` : number of internal polygons;
10. `Vertex & vertex(unsigned i)` : reference to the `i`-th vertex;
11. `Edge   & edge  (unsigned i)` : reference to the `i`-th edge;
12. `Poly   & poly  (unsigned i)` : reference to `i`-th polygon;
13. `unsigned local_number(Vertex & v)` : location of vertex `v`;
14. `unsigned local_number(Edge   & e)` : location of edge `e`;
15. `unsigned local_number(Poly   & p)` : location of polygon `p`.

Methods 1–3, 4–6 and 7–9 return the total number of instances of type Vertex, Edge and Poly and the number of internal and boundary instances in the mesh data set.

Methods 10–12 return the reference to the `i`-th vertex/edge/polygon instance in the mesh data set. Methods 13–15 accept a reference to an instance of a given type and return its internal location, that is the location in the mesh data set container. The internal location may be used as the integer identifier (global number) of that instance. As it is usual in C and C++ programming, numbering starts from `0`.

Finally the method

```
void bbox(double & xmin, double & ymin, double & xmax, double & ymax)
```

returns the coordinates of the bottom-left and the top-right extrema defining the bounding box of the mesh.

When the application program starts its execution, the mesh data set must be initialized by invoking one of the available mesh builder methods that are provided by the library either to be called as independent methods or to implement application constructors of the project class Mesh:

—`tensor_mesh` : builds a regular mesh of triangles or quadrilaterals inside a boxed frame;

—`std_tensor_mesh` : builds a regular mesh of triangles or quadrilaterals inside a unit boxed frame;

—`map_mesh` : builds a regular mesh of triangles or quadrilaterals inside a boxed frame remapped onto a four-side domain of general shape;

—`read_map_mesh` : initializes a mesh from an input file description of a regular mesh of triangles or quadrilaterals inside a boxed frame remapped onto a four-side domain of general shape;

—`build_mesh` : initializes a mesh from an input description of an unstructured mesh by using a list of vertex coordinates, a polygon-vertex connectivity, and optionally a list of edge-vertex connectivity;

—`read_mesh` : performs the same action as the previous method, but the input mesh is read from external files in the output format coming from the mesh generator `TRIANGLE`, see Shewchuk [1996].

The first four methods build a regular mesh of triangles – three different orientations are possible – or quadrilaterals starting from a *structured* description of the domain triangulation. This description is internally generated for the first three methods, and given as an input data file for the fourth one. The method `map_mesh` remaps the computational domain onto a four-side domain of general shape by a project coordinate transformation function.

The method `build_mesh` initializes the mesh data set starting from an *unstructured* description of the initial triangulation, provided by the project application as lists of coordinates and connectivity. The method `read_mesh` performs the mesh data set initialization starting from an input data files in the output format of the mesh generator `TRIANGLE`. For both methods, the list of vertex coordinates and the list of polygon–vertex connectivity are mandatory; a list of edges can be optionally given in input. The data set of edges is initialized by using such a list whenever edge–vertex connectivity is available. It is worth noticing that the external inputs are absolutely not restricted to the output file format of

TRIANGLE. Any file format can be used by programming a simple reader function and combining it with the method build_mesh.

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 argument list of the library class p2_common.

If an external mesh is given by input files, the object markers are simply read. If the mesh is generated internally , a set of markers following an internal convention – see the documentation of P2MESH – is automatically created. These internal markers indicate the location on the mesh of the entities that have been internally generated. At run-time mesh instantiation, the mesh builder method called by the application program performs an internal loop over all the vertices, edges and triangles of the mesh data set, and for any entity invokes the specific marker processing function if required. No action occurs if the marker processing function pointer is set to NULL.

4.5 Iterators

An iterator in P2MESH is a particular object which allows to iterate over the vertices, edges, and polygons of the mesh data set in a very effective and transparent way. The use of iterators does not make visible the actual processing of the items iterated by the program.

Since P2MESH has been designed and implemented using as much as possible the technology of the STL, two families of STL-like iterator types – for resp. objects and constant objects – are naturally derived

—vertex_iterator, edge_iterator, poly_iterator;

—vertex_const_iterator, edge_const_iterator, poly_const_iterator.

Iterators are defined as public types in the scope of the class Mesh, that is the scope operator must be used, e.g. Mesh::vertex_iterator. Some methods are provided to return “smart pointers” to particular items in the set of vertices, edges, and polygons. For example, in the case of vertices we have

—vertex_begin() : returns a smart pointer to the first item in the set of vertices,

—vertex_end() : returns a smart pointer to the one past-the-end item in the set of vertices.

Similar action is performed on the subset of internal vertices by the methods

—ivertex_begin(), ivertex_end()

and the subset of boundary vertices by

—bvertex_begin(), bvertex_end().

For edges and polygons, P2MESH makes available analogous member methods; it is simply sufficient to substitute the string vertex with edge or poly in the previous names.

The software library P2MESH contains also an iterator template, named Iterator<T> – CIterator<T> for the constant iterator type. The dummy label T indicates one of the three project classes Vertex, Edge, Poly.

The iterator template is implemented by using the iterator types defined in the STL and a more user friendly syntax is allowed.

The iterator constructor `Iterator<T> (Mesh & M, unsigned f=0)` is supported, where the reference `M` stands for the mesh data set and the flag `f=0,1,2` the iteration range – respectively all, boundary and internal items. `M` and `f` can also be set by using the public method `void set_loop(Mesh & M, unsigned f)`. An iteration loop can be specified by using

—`void begin()` : sets iterator to the first item;

—`bool end_of_loop()` : returns true if all items were visited by the iteration;

—`Iterator<T> const & operator ++ ()` : advances the iterator to the next item;

—`T const * operator () () const` and `T * operator () ()` : returns the current item pointed by the iterator.

As an example, the following code fragment illustrates how iterators can be instantiated and used in an application program.

```
Mesh mesh ;                            // creates a mesh object
Iterator<Edge> edge_iterator ;         // creates an edge iterator
edge_iterator . set_loop( mesh, 2 ) ;  // sets loop range on internal edges
for ( edge_iterator . begin() ;        // points to the internal edge
      ! edge_iterator . end_of_loop() ; // is it the last edge?
      ++edge_iterator ) {              // advances the edge pointer
     Edge & e  = *edge_iterator ;      // reference to the current edge
     Edge * pe = &*edge_iterator ;     // pointer   to the current edge
}
```

A special macro, named `foreach`, is also available by default and can be set undefined on request. It is defined by the following preprocessor statement

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

which allows a very short and effective definition of loops

```
Mesh mesh ;                            // creates a mesh object
Iterator<Edge> edge_iterator ;         // creates an edge iterator
edge_iterator . set_loop( mesh, 2 ) ;  // sets loop range on internal edges
foreach( edge_iterator ) {             // for each items
     Edge & e  = *edge_iterator ;      // reference to the current edge
     Edge * pe = &*edge_iterator ;     // pointer   to the current edge
}
```

4.6 Memory requirements and computational costs

A performance criterium has been taken into account in the design phase of the P2MESH development process. That is, a P2MESH-based software should show a “reasonable” behavior from the viewpoint of computational costs and memory storage requirements.

In order to achieve this goal, P2MESH guarantees that the first-neighbour topology of the mesh is generally available at the cost of a single pointer dereferentiation plus at most one comparison. The latter additional cost is needed only for retrieving the edges adjacent an edge and the polygons adjacent a polygon. P2MESH always stores four pointers for each edge and $2\times$`size` pointers for each polygon, where `size=3` for triangular meshes and 4 for quadrilateral meshes. Lists of pointers for the vertex topology are not provided by default, and stored only upon explicit request by the library user.

The total memory requirement for a general unstructured mesh of N_v vertices, N_e edges and N_p polygons, either triangles or quadrilaterals can be thus easily estimated. Since the relation $N_e = N_v + N_p - 1$ holds true, a mesh of N_v vertices and N_p polygons is represented in the P2MESH abstraction without explicit vertex topology by $4\,(N_v - 1) + 2\,(\texttt{size}{+}2)\,N_p$ pointers. In addition, P2MESH stores $2\,N_v$ double precision floating point numbers (default built-in arithmetic type) for the vertex coordinates.

In the case that vertex topology is requested, the previous memory requirement must be increased by the number of pointers of each vertex to the connected vertices and to the adjacent edges and polygons. In a mesh with no vertex connectivity constraints, as is the case of a general unstructured grid, this additional cost is difficult to be estimated, but it is likely to have a great impact on the storage requirements.

Memory requirements could have been halved by storing only two pointers to edges (or, alternatively, to vertices) in each edge and size pointers to vertices (or, alternatively, to edges) in each polygon. Furtherly, when no edge topology is needed by a numerical algorithm, no pointers at all might be stored in edges. Notice that these ones are just different design choices. To this purpose, we believe that, when developing a general purpose tool, it is better to pay some redundancy in simpler cases and gain nearly constant-in-time access in a wider range of situations. Moreover, we outline that the minimum memory requirement is normally amortized when an application is developed in the spirit of P2MESH, since each geometric structure is extended to store a larger numerical data set.

All the geometrical and mapping methods require a run-time calculation. Thus, if these methods are repeatedly invoked to compute some expensive quantity such as the area of a polygon, a performance penalty may occur. In such a case, we suggest a very simple but indeed effective strategy to tune the code and get better performances [Bertolazzi and Manzini 1999b]. Calculate expensive geometric quantities at the beginning of the run and store them in their natural place – for example, the area of a polygon should be stored as an attribute of polygon objects. Then, instantiate appropriate methods to return the value of the previously estimated quantities, which override the homonymous library methods.

5. TECHNICAL DETAILS

P2MESH is a parametric mesh-manager that implements the non-procedural programming paradigm described earlier. Its implementation uses the C++ programming language and strongly exploits the basic OO mechanisms of *encapsulation*, *inheritance*, *polymorphism* and *genericity*, see [Stroustrup 1998].

C++ supports the OOP model, is easily available and allows a form of static polymorphism by compile-time type parameterization via the template mechanism. Practical experience with the C++ template technology shows that high quality implementations can achieve a computational cost as much as the best FORTRAN implementations, see Veldhuizen [1999].

As for data containers, the implementation of P2MESH relies on the STL. This is not a constraint on portability, since the STL is part of the standard ANSI definition of C++ and must be available in any ANSI compliant compiler.

The software library P2MESH is written in an STL-like programming style, so that the library does not require any installation procedure and no library files with the usual Unix extension “.a” must be precompiled and linked. The whole library consists of a **single**

header file which has to be simply included at the beginning of each source file using P2MESH facilities.

P2MESH-based numerical codes have been successfully run on the following compilers and platforms:

Compiler	Version	Operating System
egcs	1.1	Linux 2.2.12
egcs	1.1	Solaris SunOS 5.7
Sun Workshop Compiler C++	5.0	Solaris SunOS 5.7
Metrowerks Codewarrior	4.0	MacOS 8.5
Digital C++	6.1	Digital UNIX V4.0D

6. AN APPLICATION EXAMPLE

This section illustrates an application of P2MESH in defining data types for a Poisson solver which uses linear conforming finite elements. For sake of simplicity, only Dirichlet boundary conditions are considered. The full source code is detailed in the appendix. Other examples are part of the P2MESH distribution package and completely explained in [Bertolazzi and Manzini 1999c].

Line 1 includes the library header file. Lines 2–8 include the C interface to the FORTRAN library LAPACK [Anderson et al. 1995].

The project class names are declared in lines 9–13.

The class Common – lines 16–19 – contains the static constant `ndof`, which represents the number of degrees-of-freedom and is set to 3.

The class Vertex is defined in lines 20–24, the class Edge in line 25, the class Triangle in lines 26–37, and the class Mesh in line 38.

The class Vertex contains two public methods, which respectively return the node number – or equation number – and whether the current vertex is a boundary vertex. The class Triangle contains two similar methods to interface the corresponding vertex methods and two member functions that compute the integrals in the FE variational formulation.

The bodies of the class Edge and Mesh are empty. Objects of these types contain only the topological and geometrical information accessible through the public interfaces inherited from the corresponding classes in P2MESH.

The class Elliptic_Solver is defined in lines 39–49 and contains a constructor method and the member function `Elliptic_Solver::Solve()`. The constructor instantiates a triangle-based regular mesh inside a unit frame box by calling the mesh builder method `std_tensor_mesh`.

The member function `Elliptic_Solver::Solve()` practically embodies the solver implementation following the guidelines described in [Johnson 1990]. The arrays for LAPACK are instantiated in lines 71-76. The global stiffness matrix and the load vector are built in lines 77-89 by looping over the elements and by assembling their local contributions. In lines 90-96 boundary conditions are set by looping on boundary vertices. LAPACK routines are called in lines 97-100 and the approximate solution is dumped using the MTV format in line 101-114.

Remark that this example program is not tuned for performance. However, it illustrates how the usage of P2MESH can help in implementing a PDE solver.

7. FINAL REMARKS AND FUTURE WORKS

In this paper we presented the first release of a long-term project which aims to develop generic software for PDE solvers on unstructured grids. An enhanced version with a dynamic memory management and with mesh refinement primitives is in progress. We are also planning a `3-D` extensions of `P2MESH`.

ACKNOWLEDGMENTS

We would like to thank 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 for the interest they manifested about `P2MESH`, and for the many discussions and suggestions that have lead to significant improvements in the software and have greatly helped us in improving the layout, notation and general readability of the paper. Finally, we would like to address special thanks to Prof. Bruce Simpson, Dr. J.-Daniel Boissonat and the team of the project Prisme at INRIA, Sophia-Antipolis, France, for the opportunity of the first official presentation of the work.

APPENDIX

```
# include "p2mesh.hh"

extern "C" { // interface to LAPACK
  void dgetrf_( int const & M, int const & N, double * A, int const & LDA,
                int * IPIV, int & INFO ) ;

  void dgetrs_( char const & TRANS, int const & N, int const & NHRS,
                double * A, int const & LDA, int * IPIV, double * B,
                int const & LDB, int & INFO ) ;
}

class Common ;
class Vertex ;
class Edge ;
class Triangle ;
class Mesh ;

class Elliptic_Solver ;

typedef double (*pFun)(const double, const double) ;

class Common : public p2_common<Vertex,Edge,Triangle,Mesh> {
protected:
  static const unsigned ndof = 3 ; // number of degrees-of-freedom
} ;

class Vertex : public p2_vertex<Common> {
public:
  unsigned vertex_id( Mesh const & mesh ) const ;
  bool     is_on_boundary( Mesh const & mesh ) const ;
} ;

class Edge : public p2_edge<Common> { } ;
class Triangle : public p2_poly<Common> {
private:
  unsigned opps_edge(unsigned const iv) const { return (iv+1)%n_vertex() ; }
  unsigned next_edge(unsigned const iv) const { return iv ; }
  unsigned prev_edge(unsigned const iv) const
    { return (iv+n_vertex()-1)%n_vertex() ; }
public:
  double   load_vector( unsigned i, pFun f) ;
  double   stiffness_matrix( unsigned i, unsigned j ) ;
  unsigned vertex_id( unsigned iv, Mesh const & mesh ) const ;
  bool     is_vertex_on_boundary( unsigned iv, Mesh const & mesh ) const ;
} ;

class Mesh : public p2_mesh<Common> {} ;

class Elliptic_Solver : public Common {
private:
  Mesh mesh ;
  pFun f, g ;
  void set_zero( unsigned const n, double * array ) ;
public:
  Elliptic_Solver( unsigned nx, unsigned ny, pFun _f, pFun _g ) : f(_f), g(_g)
  { mesh . std_tensor_mesh(nx, ny, NULL, NULL, NULL) ; }
  ~Elliptic_Solver() {} ;
  void Solve() ;
} ;

unsigned Vertex::vertex_id( Mesh const & mesh ) const
{ return mesh . local_number( *this ) ; }

bool Vertex::is_on_boundary( Mesh const & mesh ) const
{ return mesh . local_number( *this ) < mesh . n_bvertex() ; }

double Triangle::stiffness_matrix( unsigned iv, unsigned jv ) {
  unsigned ie = opps_edge( iv ) ;
  unsigned je = opps_edge( jv ) ;
  return 0.25 * ( nx(ie) * nx(je) + ny(ie) * ny(je) ) / area() ;
}

double Triangle::load_vector( unsigned iv, pFun f ) {
  unsigned ne = next_edge( iv ) ;
  unsigned pe = prev_edge( iv ) ;
  return ( f( xm(ne), ym(ne) ) + f( xm(pe), ym(pe) ) ) * area() / 6.0 ;
}

unsigned Triangle::vertex_id( unsigned iv, Mesh const & mesh ) const
{ return vertex(iv) . vertex_id( mesh ) ; }

bool Triangle::is_vertex_on_boundary( unsigned iv, Mesh const & mesh ) const
{ return vertex(iv) . is_on_boundary( mesh ) ; }

void Elliptic_Solver::set_zero( unsigned const n, double * array )
{ for( unsigned i=0; i<n; ++i) array[i] = 0.0 ; }

void Elliptic_Solver::Solve() {
  // arrays for LAPACK
  const int ndim = 10 ;
  double mat[ ndim ][ ndim ], rhs[ ndim ] ;
  int    ipiv[ ndim ], info ;

  set_zero( ndim*ndim, (double*)mat ) ;
  set_zero( ndim, rhs ) ;

  // build global stiffness matrix and rhs
  Iterator<Triangle> triangle(mesh) ;
  foreach( triangle ) {

    for( unsigned iv=0; iv<ndof; ++iv ) {
      if ( triangle -> is_vertex_on_boundary( iv, mesh ) ) continue ;

      unsigned i = triangle -> vertex_id  ( iv, mesh ) ;
      rhs[i]    += triangle -> load_vector( iv, f ) ;

      for( unsigned jv=0; jv<ndof; ++jv ) {
        unsigned j = triangle -> vertex_id( jv, mesh ) ;
        mat[i][j] += triangle -> stiffness_matrix( iv, jv ) ;
      }
    }
  }

  // setup g(x,y) on boundary vertices
  Iterator<Vertex> vertex( mesh, 1 ) ; // 1 = Boundary_Items ;
  foreach( vertex ) {
    unsigned i = vertex -> vertex_id(mesh) ;
    mat[i][i]  = 1.0 ;
    rhs[i]     = g( vertex -> x(), vertex -> y() ) ;
  }

  // solve the system using LAPACK
  double * const sol = rhs ;
  dgetrf_( ndim, ndim, (double*)mat, ndim, ipiv, info ) ;
  dgetrs_( ’T’, ndim, 1, (double*)mat, ndim, ipiv, sol, ndim, info ) ;

  // save results (for plotmtv)
  ofstream file("p1.mtv") ;
  file << "$ DATA=CONTCURVE\n%contstyle=2" << endl ;
  foreach( triangle ) {
    for ( unsigned i=0 ; i < triangle -> n_vertex() ; ++i ) {
      unsigned v = triangle -> vertex( i ) . vertex_id( mesh ) ;
      file << triangle -> x(i) << " " << triangle -> y(i) << " "
           << sol[v] << endl ;
    }
    file << endl ;
  }
  file << "$ END" << endl ;
  file . close() ;

}
```

```
static double f(const double x, const double y) { return -4.0 ; }
static double g(const double x, const double y) { return x*x+y*y ; }

int main() {

  cout << "start computation..." ; cout . flush() ;
  Elliptic_Solver solver( 10,10, f,g ) ;
  solver . Solve() ;
  cout << "done"  << endl ;

}
```

REFERENCES

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1995. *LAPACK Users' Guide Second Edition*. SIAM. http://www.netlib.org/lapack/.

BADER, G. AND BERTI, G. 1998. Design principles of reusable software components for the numerical solution of PDE problems. In W. HACKBUSCH AND G. WITTUM Eds., *Concepts of Numerical Software* (Braunschweig, 1998).

BARTON, J. J. AND NACKMAN, L. R. 1994. *Scientific and engineering C++*. Addison-Wesley.

BASTIAN, P., BIRKEN, K., JOHANNSEN, K., LANG, S., NEUSS, N., RENTZ-REICHERT, H., AND WIENERS, C. 1997. UG - A flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science 1*, 27–40. http://cox.iwr.uni-heidelberg.de/˜ug/.

BECK, R., ERDMANN, B., AND ROITZSCH, R. 1995. KASKADE 3.0 an object-oriented adaptive finite element code. Technical Report TR–95–4, Konrad–Zuse–Zentrum für Informationstecknik, Berlin. http://www.zib.de/SciSoft/kaskade/.

BERTOLAZZI, E. AND MANZINI, G. 1999a. The kernel of P2MESH. Technical Report IAN–1166, IAN – CNR.

BERTOLAZZI, E. AND MANZINI, G. 1999b. P2MESH: Programmer's manual. Technical Report IAN–1164, IAN – CNR.

BERTOLAZZI, E. AND MANZINI, G. 1999c. P2MESH: Programming finite element and finite volume methods. Technical Report IAN–1165, IAN – CNR.

BERZINS, M., FAIRLIE, R., PENNINGTON, S. V., WARE, J. M., AND SCALES, L. E. 1998. SPRINT2D: adaptive software for PDEs. *ACM Transactions on Mathematical Software 24*, 4, 475–499. http://www.scs.leeds.ac.uk/cpde/software.html.

BRUASET, A. M. AND LANGTANGEN, H. P. 1996. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. DAEHLEN AND A. TVEITO Eds., *Numerical Methods and Software Tools in Industrial Mathematics*. Birkauser. http://www.nobjects.com/Diffpack.

CHRISTOFIDES, N. 1975. *Graph Theory. An Algorithmic Approach*. Academic Press.

CIARLET, P. 1987. *The Finite Element Methods for Elliptic Problems*. North-Holland.

COPLIEN, J. O. 1992. *Advanced C++. Programming styles and idioms*. Addison-Wesley.

DUBOIS-PÈLERIN, Y. AND PEGON, P. 1997. Improving modularity in object-oriented finite element programming. *Communications in Numerical Methods in Engineering 13*, 193–198.

DUBOIS-PÈLERIN, Y. AND PEGON, P. 1998. Linear constraints in object-oriented finite element programming. *Computer Methods in Applied Mechanics and Engineering 154*, 31–39.

EYHERAMENDY, D. AND ZIMMERMANN, T. 1998. Object-oriented finite elements III. Theory and application of automatic programming. *Computer Methods in Applied Mechanics and Engineering 154*, 41–68.

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

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

GEORGE, P. L. 1991. *Automatic mesh generation. Application to finite element methods*. John Wiley & Sons.

GEORGE, P. L. 1996. Automatic mesh generation and finite element computation. In P. CIARLET AND J. LIONS Eds., *Handbook of Numerical Analysis*, Volume IV. North-Holland.

GLOTH, O., VILSMEIER, R., AND HÂNEL, D. 1998. An object oriented framework for finite volume applications. SciTools'98 - International Workshop on Modern Software Tools for Scientific Computing, SINTEF, `http://www.oslo.sintef.no/SciTools98/`.

HIRSCH, C. 1990. *Numerical Computation of Internal and External Flows*. J. Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England.

HOARE, C. A. R. 1962. Quicksort. *Computing Journal 5*, 10–15.

JOHNSON, C. 1990. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press.

KNUTH, D. E. 1998. *The Art of Computer Programming - Sorting and Searching* (second ed.), Volume 3. Addison-Wesley.

MCKENNA, F. T. 1997. *Object-oriented finite element programming: frameworks for analysis, algorithms and parallel computing*. Ph. D. thesis, University of California, Berkeley.

MUSSER, D. AND SAINI, A. 1996. *STL tutorial & reference guide: C++ programming with the Standard Template Library*. Addison-Wesley.

MUSSER, D. R. AND STEPANOV, A. A. 1994. Algorithm-oriented generic libraries. *Software–practice and experience 24*, 623–642.

SELMIN, V. AND FORMAGGIA, L. 1996. Unified construction of finite element and finite volume discretizations for compressible flows. *International Journal for Numerical Methods in Engineering 39*, 1–32.

SHEWCHUK, J. R. 1996. A two-dimensional quality mesh generator and Delaunay triangulator. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/quake/public/www/triangle.%html`.

SIMPSON, R. 1997. A data modeling abstraction for describing triangular mesh algorithms. *BIT 37(1)*, 138–163.

STROUSTRUP, B. 1998. *The C++ Programming Language (third edition)*. Addison-Wesley.

VANKEIRSBILCK, P. AND NELISSEN, G. 1994. ELEMD: An object-oriented software system for grid elements with multiple discretizations for solving PDEs. In S. EDITOR Ed., *Second Annual Object-Oriented Numerics Conference* (Sunriver, Oregon, 1994). SIAM.

VELDHUIZEN, T. L. 1995a. Expression templates. *C++ Report 7*, 5, 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

VELDHUIZEN, T. L. 1995b. Using C++ template metaprograms. *C++ Report 7*, 4, 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

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

ZEGLINSKY, G. W. AND HAN, R. P. S. 1994. Object-oriented matrix classes for use in a finite element code using C++. *International Journal for Numerical Methods in Engineering 37*, 3921–3937.

ZIMMERMANN, T., DUBOIS-PÈLERIN, Y., AND BOMME, P. 1992. Object-oriented finite element programming: I. Governing principles. *Computer Methods in Applied Mechanics and Engineering 98*, 291–303.