ISTITUTO
DI
ANALISI NUMERICA

del

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

PAVIA
1999

PUBBLICAZIONI

N. 1165

Enrico Bertolazzi, Gianmarco Manzini

P2MESH: Programming Finite Element and Finite Volume Methods

P2MESH: Programming Finite Element and Finite Volume Methods

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. Four different examples show the practical application of P2MESH to the development of numerical solvers for PDE problems.

(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

The P2MESH software library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

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 The model problem **3**

2 FE and FV discretizations **5**

2.1 Basic ideas of Finite Element Methods 5

2.2 Basic ideas of Finite Volume Methods 8

3 A FE solver for the Poisson problem **11**

4 A $\mathcal{P}_2$ conforming solution **12**

5 A $\mathcal{Q}_2$ conforming solution **28**

6 A FV solver for the compressible Euler equation **44**

7 A cell centered finite volume solution **45**

8 A vertex centered finite volume solution **61**

A The file “eu.hh” **77**

B The file “eu.cc” **77**

1 The model problem

Let Ω be an open subset of $\mathbb{R}^n$ with boundary $\partial\Omega$ and T a real positive constant. The mathematical form of a typical time dependent problem is

$$\begin{cases} \dfrac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F} = \mathbf{Q}, & \mathbf{x} \in \Omega, \quad t \in [0,\, T] \\ \text{boundary conditions for} & \mathbf{x} \in \partial\Omega, \quad t \in [0,\, T] \\ \mathbf{U} \text{ assigned at } t = 0 \text{ for} & \mathbf{x} \in \Omega, \end{cases} \tag{1}$$

where $\mathbf{U} = \mathbf{U}(t, \mathbf{x})$ is the vector of unknowns, $\mathbf{Q} = \mathbf{Q}(t, \mathbf{x})$ is a given source term, and $\mathbf{F} = \mathbf{F}(t, \mathbf{x}, \mathbf{U}, \nabla\mathbf{U}, \ldots)$ takes into account the dependence on $\mathbf{U}$ and its spatial derivatives. The mathematical form of a stationary problem is

$$\begin{cases} \nabla \cdot \mathbf{F} = \mathbf{Q}, & \mathbf{x} \in \Omega, \\ \text{boundary conditions for} & \mathbf{x} \in \partial\Omega, \end{cases} \tag{2}$$

with similar definitions for $\mathbf{U}$, $\mathbf{Q}$, $\mathbf{F}$.

A suitable set of boundary conditions and initial solutions must be provided in order to have a well-defined mathematical problem. The *conservative*, or sometimes called *divergence* form, has been preferred because it is a natural starting point for Finite Volume (FV) methods. These ideas apply to FV methods as well as to Finite Element (FE) ones.

Both FV and FE discretizations are based on a *mesh triangulation*, that is on the partitioning of the computational domain Ω, in some basic geometrical entities (usually called cells, elements or control volumes), such as triangles, quadrilaterals in 2-D, or tetrahedrons, prisms, in 3-D. Managing such entities may be trivial in simple situations, for instance when one deals with a structured rectangular mesh on a simple domain (such as a square), but it is not at all evident when an unstructured mesh on a general shaped domain is considered.

A PDE solver must be capable to handle all the topological and geometrical information required by the numerical algorithm. Such a solver generally manages information whose nature depends on both the applications and the numerics. The main difference in the implementation of different algorithms relies in the *data structures* involved and the way these latter ones are *manipulated*.

Basically, all relevant information, such as geometrical quantities, physical unknowns and auxiliary dependent variables, can be logically associated to different geometri-

cal entities, which intuitively correspond in the 2-D case to a Vertex, an Edge, or a Polygon in the mesh. More complex mesh-based data structures can be built as the completion of a set of suitably parameterized geometric containers and the mesh itself is a container for instances of these data structures.

In this framework, the design of a PDE solver requires the careful specification of the basic data types and their functionalities. At the highest level of abstraction, a very general solution approach is truly independent of the details of the problem to be solved and of the discretization method to be applied.

For example, one considers a geometrical type such as an Edge, and wants to extend it into a more complex data type, in such a way that certain basic properties of the basic type continue to hold for the extended type. The application-dependent extended Edge will transparently manage the underlying geometric mesh and will contain all the data and functionalities required by the problem and the approximation algorithm under consideration.

In this respect, OOP techniques turn out to be quite effective. Actually, in a procedural programming model, the data structures are manipulated by external procedures that take them as input/output arguments. Instead, in non-procedural OOP models the *different functionalities which operate upon different data are themselves part of the data specification*, see [7].

OOP techniques are versatile in the description of data containers and help to isolate the data structure design from the implementation of the application program.

This work mainly deals with the following issue of scientific programming:

> *"how to design a PDE solver for unstructured mesh computation in terms of generic parameterized containers for mesh-based data structures and related functionalities".*

The parameterization of a container is mainly achievable by these three strategies:

1. *encapsulation* of the user defined data;
2. *inheritance* from abstract base containers;
3. *direct parameterization*.

The latter one makes possible very effective implementations, because it dramatically reduces the number of pointer dereferences and the function-call overheads. Remark,

also, that the first technique is supported by any procedural language and the second one, which needs the mechanism of inheritance, is present in almost all object-oriented languages. Direct parameterization, instead, is the most recent developed technique and just few programming languages support it (e.g. ADA, C++ and EIFFEL).

2 FE and FV discretizations

The exact solution of the problems stated in equations (1-2) is usually unknown and impossible to obtain in very general situations by analytical methods. Hence, these problems must be reformulated in a suitable *discrete* form, which allows computations.

The section illustrates some basic features of the Finite Element and Finite Volume methods, focusing on those computational aspects related to the management of data structures in a numerical solver.

2.1 Basic ideas of Finite Element Methods

In literature there are many papers and textbooks devoted to the presentation of the theoretical aspects of the finite element methods, which discuss the properties of the above formulations, show the way to construct suitable discrete approximations and analyze convergence in terms of error estimates. A general but also detailed presentation of these issues can be found, for example, in [1].

The present section reviews some essential issues on FE methods by an application to the homogeneous Laplace equation. The problem

$$\begin{cases} -\Delta u = f, & \text{in } \Omega \\ u = 0, & \text{on } \partial\Omega \end{cases}$$

can be considered in the framework of (2) with $\mathbf{U} = u$, $\mathbf{F}(u) = -\nabla u$ and $\mathbf{Q} = f$. The starting point of every FE method is the “variational formulation” of the problem.

$$\begin{cases} \text{Find } u \text{ in a space of admissible functions } V \text{ such that} \\ a(u,\, v) = L(v) \ \text{ for all } v \in V, \end{cases} \tag{3}$$

where $u \in V$ is the weak solution. In the particular case of the homogeneous Laplace problem, $V = H_0^1(\Omega)$ indicates the standard Sobolev space of square-integrable functions with null trace on the boundary and whose first derivatives are all square-integrable functions. The bilinear form $a(\cdot,\, \cdot) : V \times V \to \mathbb{R}$ and the linear functional $L(\cdot) : V \to \mathbb{R}$ are

$$a(u,\, v) = \int_\Omega \nabla u \cdot \nabla v, \qquad L(v) = \int_\Omega f\, v. \tag{4}$$

The discrete form of the variational formulation derived from (3) and suitable for computation is formally given by

$$\begin{cases} \text{Find } u_h \text{ in a space of admissible functions } V_h \text{ such that} \\ a_h(u_h,\, v_h) = L_h(v_h) \ \text{ for all } v_h \in V_h, \end{cases} \tag{5}$$

where u_h is the approximation of u in the finite-dimensional space V_h, which is, in the case of conforming finite elements, a subspace of the Hilbert space V. The bilinear form $a_h(\cdot,\cdot) : V_h \times V_h \to \mathbb{R}$ and the functional $L_h(\cdot) : V_h \to \mathbb{R}$ are suitable approximations of the bilinear form $a(\cdot,\, \cdot)$ and the linear functional $L(\cdot)$ introduced in (4), respectively. The simplest choice is $a_h(\cdot,\cdot) = a(\cdot,\cdot)$ and $L_h(\cdot) = L(\cdot)$, but more complex ones are also possible, depending on different choices for V_h and also involving approximations by suitable quadrature rules of the integrals in equation (5).

Roughly speaking, a FE method requires the definition of a finite-dimensional subspace V_h which should be convenient from both a theoretical viewpoint, because it preserves some useful properties of the exact solution space V, and from a computational viewpoint, because it allows an easy estimation of the discrete approximation $u_h \in V_h$ to u.

The finite-dimensional space of admissible functions V_h is essentially built by piecewise polynomials defined on a *mesh* or *triangulation*, usually denoted by $\mathcal{T}_h$, of the domain Ω_h, which is an approximation of the domain Ω. A triangulation is a union of a suitable set of *elements* K and is generally demanded to satisfy some regularity constraints, which affect both the more practical implementation aspects and the eventual underlying theoretical analysis.

The space V_h usually consists in functions whose restriction to any element $K \in \mathcal{T}_h$ is a polynomial of an assigned order. Some regularity conditions are usually satisfied by the functions in V_h, and sometimes also by their derivatives, such as the global continuity on the closure of the domain $\overline{\Omega}_h$).

The key point in the construction of the space V_h is that there exists a set of basis functions $\{v_i\}$ having a small support. This fact enormously simplifies the computation of the integrals in $a_h(\cdot,\cdot)$ and $L_h(\cdot)$. Thus, the approximate solution u_h is given by the linear combination of the functions $\{v_i\}$, that is

$$u_h = \sum_{i=1}^{N} u_i v_i. \tag{6}$$

The N terms $\mathbf{u} = \{u_i\}$ are generally called the "degrees of freedom". They are computed by solving the linear system that arises from relation (5) for all the basis functions $\{v_i\}$.

A particular choice of the basis functions $\{v_i\}$ strongly affects the meaning of the degrees of freedom; for example, in Lagrange-type finite elements, the degrees of freedom approximate the values of the solution at a given set of nodes within the element; in Hermite-type finite elements the degrees of freedom approximate also the values of the directional derivatives at some given locations in K.

Thus, a finite element method formally implies a suitable choice for the triplet (K, Σ_K, P_K), where Σ_K is the set of degrees of freedom and P_K the space of polynomials defined on the generic geometrical element K. Different choices of this triplet correspond to different finite element methods and are equivalent to the specification of V_h once a triangulation $\mathcal{T}_h$ has been given.

Finally, substitution of (6) in (5) yields the linear algebraic problem

$$\mathbf{A}\mathbf{u} = \mathbf{b},$$

where the matrix $\mathbf{A}$ is usually called the "stiffness matrix" and the right-hand-side $\mathbf{b}$ the "load vector" (the terminology comes from elasticity problems). In a standard implementation, the stiffness matrix and the load vector are assembled from local contributions estimated on every mesh element, i.e.

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

A typical implementation is based on a loop over all the elements of the triangulation. Hence, it is very natural to utilize element-based data structures and access their values for the current element instance and its neighbors. Essential boundary condition,

dependent on the physical problem and the variational formulation, are usually considered at this stage. Finally, a library solver for linear problems is invoked, the degrees of freedom of u_i are computed and the approximate solution u_h can be post-processed.

2.2 Basic ideas of Finite Volume Methods

The FV method basically consists in producing an appropriate discretization of a set of conservation laws expressing, in an equivalent integral way, the original problem (1-2).

Methods based upon the FV spatial discretization have become very popular during the last two decades, mainly because they present some advantages on FE methods in the case of large transport terms. In particular, they are very suitable for systems of conservation laws. With respect to FE methods, where the integrals of the stiffness matrix are usually calculated by a transformation onto a reference element, the advantage of FV methods is that they are able to solve the equations directly in the computational domain. Since all the costs associated with the evaluation of the jacobian of the transformation matrix are removed, an important reduction of computational costs can occur. For the sake of exposition, the discussion in the section is restricted to time-dependent problems. The considerations presented hereafter apply also to stationary problems.

FV methods are based on a suitable *mesh* partitioning of the computational domain Ω, which can be given as a set of "control volumes" K, also called "finite volumes", or more simply "cells". Contrary to the case of FE methods, the set of volumes $\{K\}$ is not necessarily identified with $\mathcal{T}_h$. Indeed, the FV methods referred to in literature for unstructured grid computations may actually deal with two different kinds of meshes: the *primal mesh*, that is $\mathcal{T}_h$, and its *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 two meshes. Since this association is not unique, more than one dual mesh can be specified from a given primal mesh.

For the sake of exposition and without pretending of being exhaustive, the FV methods are regrouped into the two following families, depending on which mesh the control volumes are considered:

- a *cell centered* FV method takes the control volumes as elements of the primal mesh;

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

Since a number of dual meshes are possible, many variants of the cell vertex FV methods exist and are documented in literature.

The integration of equation (1) on the mesh control volumes yields the following set of integral conservation laws

$$\frac{\partial}{\partial t}\int_K \mathbf{U} + \int_K \nabla\cdot\mathbf{F} = \int_K \mathbf{Q}\,, \quad \text{for every } K \tag{7}$$

The derivative with respect to t has been taken outside the first integral under the assumption that the control volumes do not change in time. The final form, which is the most suitable for the FV discretization, is then achieved by introducing in the first integral the quantity $\overline{\mathbf{U}}_K$, which is the average value of the unknown $\mathbf{U}$ on the control volume K,

$$\overline{\mathbf{U}}_K = \frac{1}{|K|}\int_K \mathbf{U},$$

and by applying the divergence theorem to the second integral. Equations (7), then, become

$$\frac{d\overline{\mathbf{U}}_K}{dt} + \frac{1}{|K|}\int_{\partial K} \mathbf{n}_K\cdot\mathbf{F} = \frac{1}{|K|}\int_K \mathbf{Q}\,, \quad \text{for every } K \tag{8}$$

where ∂K is the boundary of control volume K and $\mathbf{n}_K$ is its outward normal vector. The discrete FV method is given by introducing the unknowns $\{\mathbf{U}_K(t^n)\}$ which approximates at time level t the cell-averaged value $\overline{\mathbf{U}}_K$ over the control volume K. A common approach consists in separating the discretization in time and space by treating them independently. This approach is historically known in literature as *the method of lines*.

Finite differences in time are usually adopted for the time derivative (first term in (8)), which produce in the simplest case 1-st order explicit or implicit time-marching schemes. Higher-order accurate and more complex approximations are produced by "predictor-corrector" and Runge-Kutta schemes.

As far as space discretization is concerned, the crucial point is that the evolution in time of the cell-averaged quantities $\{\overline{\mathbf{U}}_K\}$ depends only upon the distribution of the flux density $\mathbf{n}\cdot\mathbf{F}$ on the cell interface boundaries $\{\partial K\}$. Furthermore, the flux integral term

depends on the (pointwise) value that the solution $\mathbf{U}$ – and sometimes its gradient – takes on the control volume boundary. However, only the approximation $\{\mathbf{U}_K\}$ of the cell-averaged values $\{\overline{\mathbf{U}}_K\}$ are available during the solution process, which demands for a "recovering" step from the averages to the pointwise values.

The simplest procedure consists in taking the cell-average values as the approximation to the solution values at the centroids of the control volumes. The resulting scheme is 1-st order accurate-in-space. Higher-order accurate spatial representations are based on higher-order polynomial *reconstruction* procedures. Spurious numerical oscillations may appear when discontinuities are present. This is typical of non-linear problems but it may happen also in the simpler case of a rigid advection of an initially discontinuous solution. Numerical oscillations are avoided by requiring that some additional monotonicity constraints are satisfied by the discretization.system

No special assumptions are usually given on the regularity of the approximated solution, such as global continuity on $\bar{\Omega}$, as is the case of conforming FE methods. Thus, the solution which is locally reconstructed within any control volume K is generally discontinuous across control volume boundaries. Flux integral computation must be performed via a suitable numerical flux model, which takes into account the contributions from both sides of any internal edge and the boundary conditions set on boundary edges.

Assuming, for simplicity, that the physical flux depends only on the solution $\mathbf{U}$, the second term in equation (8) can be modelled by

$$\int_{\partial K} \mathbf{n}_K \cdot \mathbf{F}(\mathbf{U}) \approx \sum_{\tilde{K} \in \mathcal{N}(K)} \boldsymbol{\Phi}(\mathbf{U}_K, \mathbf{U}_{\tilde{K}}, \mathbf{n}_{K\tilde{K}}),$$

where $\mathcal{N}(K)$ is the set of control volumes adjacent to the control volume K, $\mathbf{n}_{K\tilde{K}}$ is the average normal along the edge shared by K and $\tilde{K} \in \mathcal{N}(K)$, and $\boldsymbol{\Phi}(\mathbf{U}_K, \mathbf{U}_{\tilde{K}}; \mathbf{n}_{K\tilde{K}})$ is the numerical flux.

An extensive number of numerical fluxes has been proposed in literature. We mention the family of central fluxes eventually corrected by an artificial dissipation term, the one of upwind fluxes, based on the (exact or approximate) solution of Riemann problems, and the one of flux-vector splitting-based fluxes.

In this framework, once one has identified the FV control volumes with the cells of the primal or the dual mesh, both the integral formulation of the original differential problem (1) and its discrete FV counterpart basically remain the same. Nevertheless, the program design and its implementation substantially differ.

In fact, when an explicit time-marching scheme is coupled with an FV method, at any time step (or any internal stage for explicit Runge-Kutta or multistage schemes) the two following operations must be performed:

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

In a cell center FV method, the control volumes are the primal mesh cells. Hence, the solver implementation may demand for 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 may demand for 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 to retrieve efficiently the information stored in objects of type, say, *edge* and *cell*; in the latter one, in objects of type *cell* and *vertex*. Other totally different implementations could be possible, of course; the ones proposed, even if rather common, just exemplify the basic ideas. Special care should also be devised to the treatment of the boundary conditions, but this issue will no be addressed for the time being.

3 A FE solver for the Poisson problem

Here we introduce the reader to the facilities provided by the `p2_mesh` software library. Since the best way to learn `P2MESH` is to use it by writing programs, some examples, commented line-by-line, will describe `P2MESH` in action. Two FE different solvers are given for the Poisson problem with non-homogeneous boundary conditions, by using conforming $\mathcal{P}_2$ and $\mathcal{Q}_2$ Lagrangian elements. Two different FV solvers are also given for the compressible Euler equations by using a cell center and a cell vertex scheme. The reader is assumed to be familiar with both theoretical and numerical issues related to these problems, since the attention in the presentation will be focused just on the implementation details concerning the usage of `P2MESH`. In all these applications, the `P2MESH` classes, which contain a complete representation of

the mesh, are inherited to define the mesh-based application classes. Due to the tutorial nature of the examples, no particular effort to produce a “good” implementation is spent in issues other than the grid representation and the mesh-based representation of the numerical algorithm and the solution. For instance, in the case of the Poisson problem, neither the sparsity nor the symmetry of the resulting linear system are at all exploited. Let $\Omega = (0,1) \times (0,1)$ be the computational domain; then, the Poisson problem with non-homogeneous Dirichlet conditions reads

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega. \end{cases}$$

The resulting algorithm follows the line described in paragraph 2.1.

4 A $\mathcal{P}_2$ conforming solution

This section describes the program contained in the file `p2_solver.cc` in the directory `examples` of the `P2MESH` distribution package.

Include the library

```
1 # include "p2mesh.hh"
```

Analysis Line `1` includes the header file of `p2mesh.hh`.

Declare the user-defined class names

```
2 class Vertex ;
3 class Edge ;
4 class Triangle ;
5 class Mesh ;
6 class Elliptic_Solver ;
```

```
7 typedef double (*pFun)(double const & x, double const & y) ;
```

Analysis Lines 2–6 declare the names of all the mesh-based project classes. A function pointer type is also declared in line 7. Declarations in lines 2–5 are mandatory by the mechanism of inheritance, while the one in line 6 could be omitted. The P2MESH library does neither require nor enforce any particular choice for the project class names. In this and following examples the names `Vertex`, `Edge`, `Triangle`, and `Mesh` are conventionally adopted for the project classes. The name `Common` is used for the common project class.

Define the class Common

```
 8 class Common : public p2_common<Vertex,Edge,Triangle,Mesh> {
 9 protected:
10   static unsigned const degree_of_freedom = 6 ;

11   static void shape(unsigned const,
12                     double const &,
13                     double const &,
14                     double &) ;

15   static void shape_grad(unsigned const,
16                          double const &,
17                          double const &,
18                          double [2]) ;
19 } ;
```

Analysis The class `Common` is publicly inherited from the library template class `p2_common`. This latter one is parameterized in line 8 by the project class names introduced in lines 2–5.

The class `Common` is a container for information, such as `enum`, `static` or `typedef` definitions, to be shared by different instances of different project classes. In very simple situations the class might also be empty. In the present case it contains the prototypes of the static functions `shape` and `shape_grad`, which are the $\mathcal{P}_2$ polynomial basis functions and their gradients.

Define the classes Vertex, Edge and Triangle

```
20 class Vertex : public p2_vertex<Common> {
21 public:
22   unsigned EqNumber    (Mesh const &) const ;
23   bool     IsOnBoundary(Mesh const &) const ;
24 } ;

25 class Edge : public p2_edge<Common> {
26 public:
27   unsigned EqNumber    (Mesh const &) const ;
28   bool     IsOnBoundary(Mesh const &) const ;
29 } ;

30 class Triangle : public p2_poly<Common> {
31 public:
32   void   eval_JJT(double[2][2], double &) const ;
33   double eval_int_f(unsigned const, pFun, double const &) const ;
34   double eval_int_grad(unsigned const, unsigned const,
35                        double const [2][2], double const &) const ;

36   unsigned EqNumber    (Mesh const &, Unsigned const) const ;
37   bool     IsOnBoundary(Mesh const &, Unsigned const) const ;
38 } ;
```

Analysis The code fragment in lines `20–38` defines the project classes `Vertex`, `Edge` and `Triangle`. The mechanism of inheritance from the base classes `p2_vertex`, `p2_edge` and `p2_poly` ensures each project class the access to the underlying mesh data representation in `P2MESH`. The base classes are moreover parameterized by the project class `Common`, in such a way that they contain also the common information of the project.

When $\mathcal{P}_2$ basis polynomials are used, the N degrees of freedom of the FE discretization may be logically assigned to mesh vertices or edge midpoints. Notice that $N = N_{vertex} + N_{edge}$. The degrees of freedom are globally enumerated (and of course uniquely identified) by an integer number running through 0 to $N-1$. The `Vertex`, `Edge` and `Triangle` types are equipped with the public method `EqNumber`, which returns the global identifier.

The public methods `Vertex::IsOnBoundary` and `Edge::IsOnBoundary` return the boolean value `true` whenever the current instance of a `Vertex` or an `Edge`

is located on the mesh boundary. The latter method is useful when the boundary conditions must be set up.

Both methods `EqNumber` and `IsOnBoundary` require the reference to the current `Mesh` object because the information they return is deduced from the rank of the instance within the mesh the instance belongs to.

The public methods `Triangle::EqNumber` and `Triangle::IsOnBoundary` take as input arguments also an unsigned integer which is the local identifier of the degree of freedom within the given triangle instance.

In the class `Triangle` the prototypes of the following three methods are also declared:

- ```
  void eval_int_f(unsigned const i,
                  pFun func,
                  double const & detJ)
  ```

  evaluates by a suitable quadrature rule on the reference element the value of the integral

  $$\iint_T f(x,y)\, v_i(x,y)\, dx\, dy = \iint_{\hat{T}} \hat{f}(s,t)\hat{v}_i(s,t)|J_F|\, ds\, dt,$$

  where f is the source term of the Poisson problem, v_i, $i = 0, 1, \ldots, 5$, are the local basis functions on the triangle T and $\hat{f}$ and $\hat{v}_i$ their counterparts defined on the reference triangle T_{ref}. A non-singular affine mapping F transforms the reference triangle T_{ref} into the actual triangle T, in such a way that the following relations hold

  $$\begin{aligned} f &= \hat{f} \circ F^{-1} \\ v_i &= \hat{v}_i \circ F^{-1}, \qquad i = 0, 1, \ldots, 5 \end{aligned}$$

  J_F is the Jacobian matrix of the mapping F and $|J_F|$ its determinant, which is constant because of the linearity of the transformation.

- ```
  double eval_int_grad(unsigned const i,
                       unsigned const j,
                       double const JJT[2][2],
                       double const & detJ) ;
  ```

computes an approximate value of

$$\iint_T \nabla v_i(x,y)\cdot\nabla v_j(x,y)\,dx\,dy = \iint_{T_{ref}} \nabla\hat{v}_i(s,t)\,J_F^{-1}J_F^{-T}\,\nabla\hat{v}_j(s,t)\,|J_F|\,ds\,dt$$

- `void eval_JJT(double JJT[2][2], double & detJ)`
  computes

$$J_F^{-1}J_F^{-T} \qquad \text{and} \qquad |J_F|$$

which are needed in the previous formulae.

Define the class Mesh

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

Analysis The class `Mesh` inherits the mesh representation from the `P2MESH` base class `p2_mesh`, which has been parameterized by the project class `Common`. No further specification is required in the current application, and the class functionalities are the ones inherited from `p2_mesh`.

Define the solver class Elliptic_Solver

```
40 class Elliptic_Solver : public Common {

41 private:
42   Mesh   mesh ;
43   double **mat ;
44   double *sol, *rhs ;

45 public:
46   Elliptic_Solver(void) {} ;
47   ~Elliptic_Solver(void) {} ;
```

```
48   void Solve(pFun, pFun, unsigned const, unsigned const) ;
49   void Save_Mtv(void) ;
50 } ;
```

Analysis The main solver class of the application is called `Elliptic_Solver`. The `P2MESH` library does not support any specific discretization method; thus, the final user must explicitly supply its implementation. The class definition is given in lines `40–50`. The solver class contains a private instance of `Mesh`, see line `42`, and the working arrays required to store the stiffness matrix, `mat`, the right-hand-side vector, `rhs`, and the solution vector `sol`, see lines `43–44`. The stiffness matrix is implemented in a rather usual style in C and C++, that is, by an array of pointers to the array of double storing the matrix rows. The statements in lines `48–49` declare the prototype of the public methods `Solve` and `Save_Mtv`. The former one implements the FE method while the latter one dumps out the approximated solution in a rather common graphic format (MTV). `Solve` takes four arguments in input: two function pointers to the functions `f` and `g`, whose type is globally declared in line `7`, and two integers `nx` and `ny`, which are used to specify the partitioning in the directions `x` and `y` of the domain Ω. The method actually builds the mesh data set, performs some local computations and assembles the global right-hand-side vector and stiffness matrix. Then, it computes the approximate solution, which is finally stored in `sol`, by solving the resulting linear system by a standard (and rather inefficient) factorization technique.

The method `Save_Mtv` saves the solution for graphical post-processing by using the "MTV" data format in a file which can be immediately visualized by the program `plotmtv`[1].

The methods of the class Common

```
51 void
52 Common::shape(unsigned const nb,
53               double const & s, double const & t,
54               double & res) {
55   switch ( nb ) {
56   case 0: res = (1-2*(s+t))*(1-(s+t)) ; break ;
57   case 1: res = 4*s*(1-(s+t))         ; break ;
```

[1] The `plotmtv` program

```
58   case 2: res = s*(2*s-1)            ; break ;
59   case 3: res = 4*s*t                ; break ;
60   case 4: res = (2*t-1)*t            ; break ;
61   case 5: res = 4*(1-(s+t))*t        ; break ;
62   }
63 }
```

Analysis The source fragment defines the local basis functions for the conforming $\mathcal{P}_2$ polynomials in the reference triangle T_{ref}. T_{ref} is the simplex $\{(s,t) \mid s,t \geq 0; s+t \leq 1\}$. The value of the nb-th local basis function at the position (s,t) is returned by the method shape in the array res. The degrees of freedom within the reference triangle T_{ref} are enumerated as shown in figure 1.

```
64 // values of gradients of bases function
65 void
66 Common::shape_grad(unsigned const nb,
67                    double const & s,
68                    double const & t,
69                    double g[2]) {
70   switch ( nb ) {
71   case 0: g[0] = 4*(s+t)-3     ; g[1] = 4*(s+t)-3     ; break ;
72   case 1: g[0] = 4 - 8*s - 4*t ; g[1] = -4*s          ; break ;
73   case 2: g[0] = 4*s-1         ; g[1] = 0             ; break ;
74   case 3: g[0] = 4*t           ; g[1] = 4*s           ; break ;
75   case 4: g[0] = 0             ; g[1] = 4*t-1         ; break ;
76   case 5: g[0] = -4*t          ; g[1] = 4 - 4*s - 8*t ; break ;
77   }
78 }
```

Analysis The source fragment defines the function shape_grad, which returns in the array g[2] the two components of the gradient of the nb-th local basis function at the point (s,t) in the reference triangle.

The methods of the class Vertex and Edge

```
79 inline
80 unsigned
```

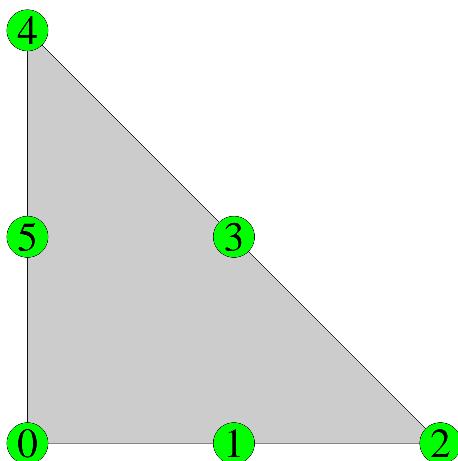

Figure 1:

```
81 Vertex::EqNumber(Mesh const & m) const
82 { return m . local_number(*this); }

83 inline
84 bool
85 Vertex::IsOnBoundary(Mesh const & m) const
86 { return m . local_number(*this) < m . n_bvertex() ; }

87 inline
88 unsigned
89 Edge::EqNumber(Mesh const & m) const
90 { return m . n_vertex() + m . local_number(*this) ; }

91 inline
92 bool
93 Edge::IsOnBoundary(Mesh const & m) const
94 { return m . local_number(*this) < m . n_bedge() ; }
```

Analysis The source fragment defines the implementation of the methods `EqNumber` and `IsOnBoundary` for `Vertex` and `Edge` type objects. Both methods make usage of the internal numbering of vertices and edges, which is returned by the function `local_number`. When `EqNumber` is invoked by a `Vertex` object, it returns the rank in the mesh of the current vertex instance; instead, when `EqNumber` is invoked

by an Edge object, it returns the rank of the current edge instance, augmented by the total number of Vertex instances in the mesh. The degrees of freedom associated to the edges of the mesh are in this way enumerated after the ones associated to the vertices. Finally, since all the boundary objects are ordered before the internal ones in the underlying mesh representation, a simple test on the rank distinguishes internal vertices and edges from the boundary ones.

The methods of the class Triangle

```
95  void
96  Triangle::eval_JJT(double JJT[2][2], double & detJ) const {
97    double iJ[2][2] ;
98    inverse_jacobian(0.0, 0.0, iJ) ;
99    JJT[0][0] = iJ[0][0]*iJ[0][0] + iJ[0][1]*iJ[0][1] ;
100   JJT[0][1] =
101   JJT[1][0] = iJ[0][0]*iJ[1][0] + iJ[0][1]*iJ[1][1] ;
102   JJT[1][1] = iJ[1][0]*iJ[1][0] + iJ[1][1]*iJ[1][1] ;

103   detJ = 1/(iJ[0][0] * iJ[1][1] - iJ[1][0] * iJ[0][1]) ;
104 }
```

Analysis The method in lines 95–104 computes the entries of the matrix $J_F^{-1} J_F^{-T}$ and the determinant $|J_F|$ of the affine mapping F. In the current implementation, the P2MESH library method inverse_jacobian is first invoked, which returns the Jacobian matrix J_F^{-1}. Then $J_F^{-1} J_F^{-T}$ and $|J_F|$ are directly evaluated. The first two arguments in inverse_jacobian would specify the local position (s, t) on the reference triangle where J_F^{-1} must be calculated. However, the Jacobian matrix of a linear transformation defined on the reference triangle is constant. Thus, these arguments are not really used, and are conventionally set to zero.

```
105 double
106 Triangle::eval_int_f(unsigned const i,
107                      pFun           func,
108                      double const & detJ) const {

109   static double s[] = { 0.5, 0.5, 0.0 } ;
110   static double t[] = { 0.0, 0.5, 0.5 } ;
```

```
111   double b ;
112   double res = 0 ;
113   for ( unsigned k = 0 ; k < 3 ; ++k ) {
114     shape(i, s[k], t[k], b) ;
115     res += func( xm(k), ym(k) ) * b ;
116   }
117   return detJ * res / 6 ;
118 }
```

Analysis This source fragment approximates the integral

$$|J_F| \iint_{T_{ref}} \hat{f}(s,t)\hat{v}_i(s,t)\,ds\,dt$$

by the edge midpoint quadrature rule:

$$\frac{|J_F|}{6}\left[\hat{f}(0.5,0)\,\hat{v}_i(0.5,0) + \hat{f}(0.5,0.5)\,\hat{v}_i(0.5,0.5) + \hat{f}(0,0.5)\,\hat{v}_i(0,0.5)\right]$$

```
119 double
120 Triangle::eval_int_grad(unsigned const i,
121                         unsigned const j,
122                         double const JJT[2][2],
123                         double const & detJ) const {

124   static double s[] = { 0.5, 0.5, 0.0 } ;
125   static double t[] = { 0.0, 0.5, 0.5 } ;

126   double gi[2], gj[2] ;

127   double res = 0 ;
128   for ( unsigned k = 0 ; k < 3 ; ++k ) {
129     shape_grad(i, s[k], t[k], gi) ;
130     shape_grad(j, s[k], t[k], gj) ;
131     res += JJT[0][0] * gi[0] * gj[0] +
132            JJT[0][1] * gi[0] * gj[1] +
133            JJT[1][0] * gi[1] * gj[0] +
134            JJT[1][1] * gi[1] * gj[1] ;
135   }
136   return detJ * res / 6 ;
137 }
```

Analysis This source fragment approximates the integral

$$|J_F| \iint_{T_{ref}} \nabla\hat{v}_i(s,t)^T (J_F^{-1} J_F^{-T}) \nabla\hat{v}_j(s,t)\, ds\, dt$$

by the edge midpoint quadrature formula:

$$\frac{|J_F|}{6} \Big[\nabla\hat{v}_i(0.5,0)^T (J_F^{-1} J_F^{-T}) \nabla\hat{v}_j(0.5,0) + \nabla\hat{v}_i(0.5,0.5)^T (J_F^{-1} J_F^{-T}) \nabla\hat{v}_j(0.5,0.5) + \nabla\hat{v}_i(0,0.5)^T (J_F^{-1} J_F^{-T}) \nabla\hat{v}_j(0,0.5) \Big]$$

```
138 unsigned
139 Triangle::EqNumber( Mesh const & m, Unsigned const loc) const {
140   if ( loc % 2 == 1 ) return edge(loc/2)   . EqNumber(m) ;
141   else                 return vertex(loc/2) . EqNumber(m) ;
142 }

143 bool
144 Triangle::IsOnBoundary( Mesh const & m, const Unsigned loc) const {
145   if ( loc % 2 == 1 ) return edge(loc/2)   . IsOnBoundary(m) ;
146   else                 return vertex(loc/2) . IsOnBoundary(m) ;
147 }
```

Analysis The methods `EqNumber` in lines `138–142` and `IsOnBoundary` in lines `143–147` takes in input the reference to the current instance of the mesh and the local (within the triangle) number of a degree of freedom. The first method returns the global number within the program application of the degree of freedom. The local numbering of the degrees of freedom is counterclockwise ordered, running through `0` to `5` and starting from a vertex, see Figure 1. In this case, the degrees of freedom associated to triangle vertices are given even local numbers, the ones associated to the triangle edges are given odd local numbers. The second method, instead, returns the boolean `true` whenever the input local number specifies a degree of freedom associated to a vertex or an edge on the boundary of the computational domain.

The solver code

```
148 void
149 Elliptic_Solver::Solve(pFun f, pFun g,
150                        unsigned const nx, unsigned const ny) {
151   unsigned i, j, k ;

152   // build the mesh
153   mesh . std_tensor_mesh( nx, ny, NULL, NULL, NULL ) ;

154   // allocate memory
155   unsigned neq  = mesh . n_vertex() + mesh . n_edge() ;
156   unsigned nnum = 2*neq + neq * neq ;
157   sol = new double [ nnum ] ;
158   mat = new double * [ neq ] ;
159   if ( sol == NULL || mat == NULL )
160     { cerr << "not enought memory" << endl ; exit(0) ; }
161   rhs    = sol + neq ;
162   mat[0] = rhs + neq ;
163   for ( i = 1 ; i < neq ; ++i ) mat[i] = mat[i-1] + neq ;

164   // clean up memory
165   for ( i = 0 ; i < nnum ; ++i ) sol[i] = 0 ;

166   // build the linear system
167   Iterator<Triangle> triangle(mesh) ;
168   foreach( triangle ) {
169     double detJ, JJT[2][2] ;
170     triangle -> eval_JJT(JJT, detJ) ;

171     for ( i = 0 ; i < degree_of_freedom ; ++i ) {
172       if ( triangle -> IsOnBoundary(mesh,i) ) continue ;
173       unsigned ig = triangle -> EqNumber(mesh,i) ;
174       rhs[ig] += triangle -> eval_int_f(i,f,detJ) ;
175       for ( j = 0 ; j < degree_of_freedom ; ++j ) {
176         unsigned jg = triangle -> EqNumber(mesh,j) ;
177         mat[ig][jg] += triangle -> eval_int_grad(i, j, JJT, detJ) ;
178       }
179     }
180   }

181   // setup boundary conditions
182   Iterator<Vertex> vertex(mesh,1) ;
183   foreach( vertex ) {
184     unsigned ig = vertex -> EqNumber(mesh) ;
```

```
185     mat[ig][ig] = 1 ;
186     rhs[ig] = g( vertex -> x(), vertex -> y() ) ;
187   }

188   Iterator<Edge> edge(mesh,1) ;
189   foreach( edge ) {
190     unsigned ig = edge -> EqNumber(mesh) ;
191     mat[ig][ig] = 1 ;
192     rhs[ig] = g( edge -> xm(), edge -> ym() ) ;
193   }

194   // copy rhs to the solution vector
195   for ( i = 0 ; i < neq ; ++i ) sol[i] = rhs[i] ;

196   // solve the linear system by modified  Gaussian Elimination
197   // without pivoting.
198   cout << "Solving a " << neq << "x" << neq << " linear system"
199        << endl ;
200   for ( i = 0 ; i < neq ; ++i ) {
201     for ( k = 0 ; k < neq ; ++k ) {
202       if ( k != i ) {
203         double bf = mat[k][i]/mat[i][i] ;
204         sol[k] -= bf * sol[i] ;
205         for ( j = i+1 ; j < neq ; ++j )
206           mat[k][j] -= bf * mat[i][j] ;
207       }
208     }
209   }
210   for ( i = 0 ; i < neq ; ++i ) sol[i] /= mat[i][i] ;
211 }
```

Analysis The source fragment in lines `148–211` implements the method `Solve`, which is actually the computational core of the application program. This part of the program is very close to a "procedural" routine, because of the sequential nature of the computations to be performed.

The resolution process can be schematically depicted in several steps as follows.

Mesh construction In line `153`, the `P2MESH` software system method `std_tensor_mesh` is invoked in order to build an unstructured mesh by a regular triangulation of the domain $\Omega = (0,1)\times(0,1)$. The mesh is composed by $2\cdot\mathtt{nx}\cdot\mathtt{ny}$ triangles. The three `NULL` entries in the statement indicates that no particular treatment is required for boundaries. In

a different application, suitable boundary conditions requiring some special treatment could be required. The simplest way to manage the situation is by the assignment of markers, whose conventional meaning is decided by the user. These entries may be given pointers to three user-defined functions, respectively for vertices, edges, and triangles, which take care of correctly specifying the boundary treatments.

Memory allocation and initialization

The source fragment in lines `154–163` allocates and initializes the arrays used in the construction and resolution of the linear system. In line `155–156` the total memory occupation in terms of double floating point numbers is determined, and then allocated in line `157`. In line `158` the memory required by the matrix row pointers is allocated. In line `161–162` the pointers to the arrays `rhs` and `mat` are initialized. In line `163` the pointers to the matrix rows are initialized, and finally in line `165` all the initial matrix and vectors values are set up to `0`.

Stiffness matrix and right-hand-side vector assembling

This part of the program basically implements the standard way the stiffness matrix and the r.h.s. vector are built. That is, a loop is performed on all the triangles of the mesh, and the local contribution to the stiffness matrix and the r.h.s. vector are first evaluated and then assembled into the global arrays. Boundary conditions are taken into account by a direct modification of the final global arrays. In line `167` the source code instantiates an iterator, called `triangle`, to be used for looping on all the triangles within the mesh. In line `168` the macro `foreach` implements a loop on the mesh triangles by using the iterator just introduced. Within the loop, the current triangle is given by the reference returned by the iterator `triangle`. In line `170` the method `triangle->eval_JJT` returns the values of

$$J_F^{-1} J_F^{-T} \qquad \text{and} \qquad |J_F|$$

which are stored in the matrix `JJF` and in the scalar `detJ`.

The `for` statement in lines `171` and `175` loops on the local degrees of freedom of the current triangle. The integers `ig` and `jg` store the global number corresponding to the local degree of freedom respectively indicated by `i` and `j`. If a degree of freedom is associated to a boundary vertex or edge, the loop is skipped because the corresponding boundary condition will be set in a subsequent part of the program.

In line `174` the local contribution to the r.h.s. vector from the integral

$$|J_F| \iint\limits_{T_{ref}} \hat{f}(s,t)\hat{v}_i(s,t)\,ds\,dt$$

is estimated and added to the array `rhs`.

In line `177` the components of the local stiffness matrix given by the integral

$$|J_F| \iint\limits_{T_{ref}} \nabla\hat{v}_i(s,t)^T (J_F^{-1} J_F^{-T}) \nabla\hat{v}_j(s,t)\,ds\,dt$$

are evaluated and added to the global stiffness matrix `mat`.

Boundary condition set up

The source fragment in lines `182`–`193` modifies the stiffness matrix and the r.h.s vector in order to take into account the boundary conditions on boundary vertices and edges.

The iterator `vertex` is first instantiated in line `182` and initialized to perform loops on boundary vertices. Within this loop, implemented in line `183` by using the macro `foreach`, the integer identifier `ig` is set to the degrees of freedom associated to the current boundary vertex. In lines `185`–`186` the stiffness matrix and the r.h.s. vector are modified in correspondence of the `ig`–th row in accord with the boundary conditions given by the function `g`.

The boundary condition for the degree of freedom associated to a boundary edge is set up in a similar way. The iterator `edge` is instantiated and initialized to loop on the boundary edges in line `188`, and then utilized within the macro `foreach` in `189` to implement the loop. In lines `190`–`192` the identifier `ig` is set to the degree of freedom associated to the current boundary edge, and the `ig`–th row of the stiffness matrix and the r.h.s vector are modified in accord with the boundary conditions given by the function `g`.

Linear system resolution

A modified Gaussian elimination algorithm [2] is implemented in lines `200`–`210` for the resolution of the linear system. The final solution is stored in the array `sol`.

Saving the computed solution

```
212 void
213 Elliptic_Solver::Save_Mtv(void) {
214   cout << "saving data file..." ;
215   cout . flush() ;
216   ofstream file("p2.mtv") ;
217   file << "$ DATA=CONTCURVE\n%contstyle=2 meshplot=true" << endl ;
218   Iterator<Triangle> ip(mesh) ;
219   foreach ( ip ) {
220     for ( unsigned nv = 0 ; nv < 3 ; ++nv ) {
221       Vertex & V = ip -> vertex(nv) ;
222       unsigned i = mesh . local_number(V) ;
223       file << V . x() << " " << V . y() << " " << sol[i] << endl ;
224     }
225     file << endl ;
226   }
227   file << "$ END" << endl ;
228   file . close() ;
229   cout << "saved" << endl ;
230 }
```

Analysis The final solution stored in the array `sol` is saved on disk in “MTV” format.

The driving program

```
231 static
232 double
233 f(double const &, double const &)
234 { return -4 ; }

235 static
236 double
237 g(double const & x, double const & y)
238 { return x*x+y*y ; }

239 int
240 main() {
241   Elliptic_Solver es ;
242   es . Solve( f, g, 8, 8) ;
```

```
243   es . Save_Mtv() ;
244 }
```

Analysis The driving program defines as static functions the right-hand-side term `f`, see lines `231–234`, and the boundary condition term `g`, see lines `235–238`. Then, it invokes the methods `Solve` and `Save_Mtv`.

In Figure 2, the final solution computed by the program is shown.

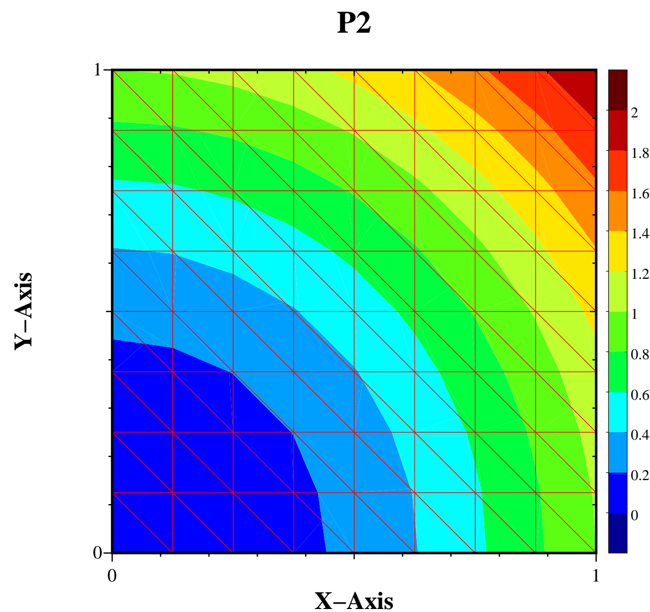

Figure 2: Linear triangle-based FE solution of the Poisson problem.

5 A $\mathcal{Q}_2$ conforming solution

This section describes the program contained in the file `q2_solver.cc` in the directory `examples` of the `P2MESH` distribution package.

Include the library

```
1 # include "p2mesh.hh"
```

Analysis The source fragment in line 1 includes the header file of P2MESH.

Declare user-defined class names

```
2 class Vertex ;
3 class Edge ;
4 class Quad ;
5 class Mesh ;
6 class Elliptic_Solver ;

7 typedef double (*pFun)(double const &, double const &) ;
```

Analysis See the comment given for the earlier example with conforming $\mathcal{P}_2$ polynomials. Notice also that in line 4 the class name `Quad` takes the place of `Triangle`.

Define the class Common

```
8  class Common : public p2_common<Vertex,Edge,Quad,Mesh,4> {
9    static double p0(double const &) ;
10   static double p1(double const &) ;
11   static double p2(double const &) ;

12   static double dp0(double const &) ;
13   static double dp1(double const &) ;
14   static double dp2(double const &) ;
15 protected:
16   static unsigned const degree_of_freedom = 9 ;

17   static void shape(unsigned const,
18                     double const &,
19                     double const &,
20                     double &) ;

21   static void shape_grad(unsigned const,
22                          double const &,
23                          double const &,
```

```
24                                 double [2]) ;
25 } ;
```

Analysis The class Common is defined by inheritance from the library class p2_common, which, on its turn, is parameterized in line 8 by using the class names declared in lines 2–5. The general considerations given for the example using the $\mathcal{P}_2$ polynomials hold here. However, notice that the value 4 is explicitly assigned to the 5-th argument in the template argument list of the class p2_common in order to build a quadrilateral-based mesh. Since the default value of this entry, which is 3, specifies a triangle-based mesh, it was not explicitly indicated in the analogous definition of the project class Common for the conforming $\mathcal{P}_2$ polynomials. The class Common contains in this case the prototype of the static functions shape and shape_grad for the basis $\mathcal{Q}_2$ polynomial functions and their gradients, and of six other "auxiliary" functions p0, p1, p2, dp0, dp1, dp2 which are useful in computing the values of the basis functions shape and shape_grad. It also contains the static integer constant degree_of_freedom, which stores the number of degrees of freedom per quadrilateral element.

Define the classes Vertex, Edge and Quad

```
26 class Vertex : public p2_vertex<Common> {
27 public:
28   unsigned EqNumber    (Mesh const &) const ;
29   bool     IsOnBoundary(Mesh const &) const ;
30 } ;

31 class Edge : public p2_edge<Common> {
32 public:
33   unsigned EqNumber    (Mesh const &) const ;
34   bool     IsOnBoundary(Mesh const &) const ;
35 } ;

36 class Quad : public p2_poly<Common> {
37   static double detJ[2][2] ;
38   static double JJT[2][2][2][2] ;
39   static double st[2] ;
40 public:
41   void   eval_JJT(void) const ;
42   double eval_int_f(unsigned const, pFun) const ;
43   double eval_int_grad(unsigned const, unsigned const) const ;
```

```
44   unsigned EqNumber    (Mesh const &, Unsigned const) const ;
45   bool     IsOnBoundary(Mesh const &, Unsigned const) const ;
46 } ;
```

Analysis The code fragment in lines 26–46 defines the project classes `Vertex`, `Edge` and `Quad` of the application. The mechanism of inheritance from the corresponding `P2MESH` base classes `p2_vertex`, `p2_edge` and `p2_poly` provides the application classes with a complete mesh representation. The base classes are moreover parameterized by the project class `Common`, in such a way that they contain also the common information of the project.

When $\mathcal{Q}_2$ base polynomials are used, the N unknowns (or degrees of freedom) of the FE discretization may be logically assigned to mesh vertices, edge midpoints or quadrilateral centroids. In this case, we have that $N = N_{vertex} + N_{edge} + N_{quads}$.

The same considerations given for the $\mathcal{P}_2$ case about `EqNumber` and `IsOnBoundary` also hold here.

In the project class `Quad`, the static arrays `detJ` and `JJT` store the values of the Jacobian at the four vertices and of the matrix $J_F^{-1} J_F^{-T}$ at the quadrature points, while the static array `st` stores the local coordinates of the quadrature nodes in the reference quadrilateral element. These arrays are implemented as static ones and consequently are shared by all the instances of the class `Quad`. This programming choice limits the memory occupation, but these arrays must be recomputed by the current quadrilateral instance before usage.

In the `Quad` class the prototypes of the following methods are also declared:

- `void eval_int_f(unsigned const i, pFun func)`

  evaluates by a suitable quadrature rule on the reference element the value of the integral

$$\iint_{Q} f(x,y)\, v_i(x,y)\, dx\, dy = \iint_{Q_{ref}} \hat{f}(s,t)\, \hat{v}_i(s,t) |J_F|\, ds\, dt$$

  where f is the source term of the Poisson problem, v_i, $i = 0, 1, \ldots 8$, are the local basis functions on the quadrilateral Q and $\hat{f}$ and $\hat{v}_i$ their counterparts defined on the reference quadrilateral Q_{ref}. A non-singular affine mapping F transforms the reference quadrilateral Q_{ref} into the actual quadrilateral Q, in such a way that the following

relations hold

$$f = \hat{f} \circ F^{-1},$$
$$v_i = \hat{v}_i \circ F^{-1}, \qquad i = 0, 1, \dots 8.$$

J_F is the Jacobian matrix of the mapping F and $|J_F|$ is its determinant. Contrary to the triangular case, $|J_F|$ is NOT constant.

- `double eval_int_grad(unsigned const i, unsigned const j)`

  evaluates an approximate value of the integral

$$\iint_T \nabla v_i(x,y)^T \nabla v_j(x,y)\, dx\, dy =$$
$$\iint_{Q_{ref}} \nabla \hat{v}_i(s,t)^T (J_F^{-1} J_F^{-T}) \nabla \hat{v}_j(s,t)\, |J_F|\, ds\, dt$$

- `void eval_JJT(void) ;`

  evaluates at the quadrature points the terms

$$J_F^{-1} J_F^{-T} \qquad \text{and} \qquad |J_F|,$$

  and stores them in the corresponding arrays.

Define the class Mesh

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

Analysis As for the $\mathcal{P}_2$ example program, the project class `Mesh` does not require any particular specification other than the public derivation from the template class `p2_mesh` parametrized by the project class `Common`. All the class functionalities are the ones inherited from `p2_mesh`.

Define the solver class Elliptic_Solver

```
48 class Elliptic_Solver : public Common {
49 private:
50   Mesh   mesh ;
51   double **mat ;
52   double *sol, *rhs ;

53 public:
54   Elliptic_Solver(void) {} ;
55   ~Elliptic_Solver(void) {} ;

56   void Solve(pFun, pFun, unsigned const, unsigned const) ;
57   void Save_Mtv(void) ;
58 } ;
```

Analysis This class definition is identical to the one of the example with $\mathcal{P}_2$ basis polynomials and the considerations given there apply equally well in this case. However, the methods have a different implementation, because the numerical algorithm is different. This issue is a practical example of how the encapsulation paradigm works.

The methods of the class Common

```
59 inline double Common::p0(double const & x) { return 0.5*(x-1)*x ; }
60 inline double Common::p1(double const & x) { return (1-x)*(1+x) ; }
61 inline double Common::p2(double const & x) { return 0.5*(x+1)*x ; }

62 inline double Common::dp0(double const & x) { return x-0.5 ; }
63 inline double Common::dp1(double const & x) { return -2*x ; }
64 inline double Common::dp2(double const & x) { return x+0.5 ; }
```

Analysis The source fragment defines some simple polynomials which are quite useful in the definition of the conforming $\mathcal{Q}_2$ basis functions.

```
65 void
66 Common::shape(unsigned const nb,
67               double const & s, double const & t,
68               double & res) {
69   switch ( nb ) {
70   case 0: res = p0(s)*p0(t) ; break ;
```

```
71    case 1: res = p1(s)*p0(t) ; break ;
72    case 2: res = p2(s)*p0(t) ; break ;
73    case 3: res = p2(s)*p1(t) ; break ;
74    case 4: res = p2(s)*p2(t) ; break ;
75    case 5: res = p1(s)*p2(t) ; break ;
76    case 6: res = p0(s)*p2(t) ; break ;
77    case 7: res = p0(s)*p1(t) ; break ;
78    case 8: res = p1(s)*p1(t) ; break ;
79    }
80  }
```

Analysis The source fragment defines the local basis functions for the conforming $\mathcal{Q}_2$ polynomials in the reference quadrilateral $Q_{ref} \equiv \{(s,t) \,|\, -1 \leq s,t \leq 1\}$. The value of the `nb`-th local basis function at the position given by the reference coordinates (s,t) is returned in `res` by invoking the method `shape` with suitable `nb`, `s` and `t` entries. The degrees of freedom within the reference quadrilateral are enumerated as shown in figure 3.

```
81  // values of gradients of bases function
82  void
83  Common::shape_grad(unsigned const nb,
84                     double const & s,
85                     double const & t,
86                     double g[2]) {
87    switch ( nb ) {
88    case 0: g[0] = dp0(s)*p0(t) ; g[1] = p0(s)*dp0(t) ; break ;
89    case 1: g[0] = dp1(s)*p0(t) ; g[1] = p1(s)*dp0(t) ; break ;
90    case 2: g[0] = dp2(s)*p0(t) ; g[1] = p2(s)*dp0(t) ; break ;
91    case 3: g[0] = dp2(s)*p1(t) ; g[1] = p2(s)*dp1(t) ; break ;
92    case 4: g[0] = dp2(s)*p2(t) ; g[1] = p2(s)*dp2(t) ; break ;
93    case 5: g[0] = dp1(s)*p2(t) ; g[1] = p1(s)*dp2(t) ; break ;
94    case 6: g[0] = dp0(s)*p2(t) ; g[1] = p0(s)*dp2(t) ; break ;
95    case 7: g[0] = dp0(s)*p1(t) ; g[1] = p0(s)*dp1(t) ; break ;
96    case 8: g[0] = dp1(s)*p1(t) ; g[1] = p1(s)*dp1(t) ; break ;
97    }
98  }
```

Analysis The source fragment defines the function `shape_grad`, which returns in the array `g[2]` the two components of the gradient of the `nb`-th local basis function at the point (s,t) in the reference quadrilateral.

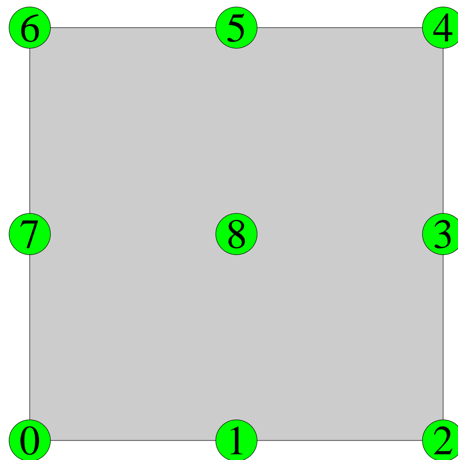

Figure 3:

The methods of the class Vertex and Edge

```
99  inline
100 unsigned
101 Vertex::EqNumber(Mesh const & m) const
102 { return m . local_number(*this) ; }

103 inline
104 bool
105 Vertex::IsOnBoundary(Mesh const & m) const
106 { return m . local_number(*this) < m . n_bvertex() ; }

107 inline
108 unsigned
109 Edge::EqNumber(Mesh const & m) const
110 { return m . n_vertex() + m . local_number(*this) ; }

111 inline
112 bool
113 Edge::IsOnBoundary(Mesh const & m) const
114 { return m . local_number(*this) < m . n_bedge() ; }
```

Analysis See the comments given for the triangular case.

The methods of the class Quad

```
115 void
116 Quad::eval_JJT() const {

117   for ( unsigned i = 0 ; i < 2 ; ++i ) {
118     for ( unsigned j  = 0 ; j < 2 ; ++j ) {
119       double iJ[2][2] ;
120       inverse_jacobian(st[i],st[j], iJ) ;
121       JJT[i][j][0][0] = iJ[0][0]*iJ[0][0] + iJ[0][1]*iJ[0][1] ;
122       JJT[i][j][0][1] =
123       JJT[i][j][1][0] = iJ[0][0]*iJ[1][0] + iJ[0][1]*iJ[1][1] ;
124       JJT[i][j][1][1] = iJ[1][0]*iJ[1][0] + iJ[1][1]*iJ[1][1] ;

125       detJ[i][j] = 1/(iJ[0][0] * iJ[1][1] - iJ[1][0] * iJ[0][1]) ;
126     }
127   }
128 }
```

Analysis This source fragment is rather similar to the one of the triangular case. However, since the Jacobian matrix and its inverse matrix are no more constant over the reference element, the library method `inverse_jacobian` is invoked with the first two arguments equal to `st[i],st[j]`, which store the local coordinates of the quadrature nodes.

```
129 double
130 Quad::eval_int_f(unsigned const i, pFun func) const {

131   double x, y, b, res = 0 ;
132   for ( unsigned ii = 0 ; ii < 2 ; ++ii ) {
133     for ( unsigned jj = 0 ; jj < 2 ; ++jj ) {
134       shape(i, st[ii], st[jj], b) ;
135       st_to_xy(st[ii], st[jj], x, y) ;
136       res += detJ[ii][jj] * func(x,y) * b ;
137     }
138   }
139   return res ;
140 }
```

Analysis The source fragment approximates the integral

$$\iint\limits_{Q_{ref}} |J_F(s,t)|\hat{f}(s,t)\hat{v}_i(s,t)\,ds\,dt$$

by the quadrature rule

$$\sum_{i,j=0,1} \omega_i\omega_j |J_F(\xi_i,\xi_j)|\hat{f}(\xi_i,\xi_j)$$

where (ξ_i, ξ_j) and ω_i are the local coordinates of the i–th quadrature point and its correspondent weight. It should be noticed that the terms ω_i in the quadrature formula here implemented are all equal to one, and thus do not explicitly appear in the implementation. Higher order formulae would have non-trivial weight values.

```
141 double
142 Quad::eval_int_grad(unsigned const i, unsigned const j) const {
143   double gi[2], gj[2], res = 0 ;
144   for ( unsigned ii = 0 ; ii < 2 ; ++ii ) {
145     for ( unsigned jj = 0 ; jj < 2 ; ++jj ) {
146       shape_grad(i, st[ii], st[jj], gi) ;
147       shape_grad(j, st[ii], st[jj], gj) ;
148       res += detJ[ii][jj] *
149              ( JJT[ii][jj][0][0] * gi[0] * gj[0] +
150                JJT[ii][jj][0][1] * gi[0] * gj[1] +
151                JJT[ii][jj][1][0] * gi[1] * gj[0] +
152                JJT[ii][jj][1][1] * gi[1] * gj[1] ) ;
153     }
154   }
155   return res ;
156 }
```

Analysis The source fragment approximates the integral

$$\iint\limits_{Q_{ref}} |J_F(s,t)|\nabla\hat{v}_i(s,t)^T(J_F^{-1}J_F^{-T})\nabla\hat{v}_j(s,t)\,ds\,dt$$

by the quadrature rule

$$\sum_{i,j=0,1} \omega_i\omega_j |J_F(\xi_i,\xi_j)|\nabla\hat{v}_i(\xi_i,\xi_j)^T(J_F^{-1}(\xi_i,\xi_j)J_F^{-T}(\xi_i,\xi_j))\nabla\hat{v}_j(\xi_i,\xi_j)$$

```
157 unsigned
158 Quad::EqNumber( Mesh const & m, Unsigned const loc) const {
159   if ( loc == 8 ) {
160     return m . n_vertex() + m . n_edge() + m . local_number(*this) ;
161   } else {
162     if ( loc % 2 == 1 ) return edge(loc/2)   . EqNumber(m) ;
163     else                return vertex(loc/2) . EqNumber(m) ;
164   }
165 }

166 bool
167 Quad::IsOnBoundary( Mesh const & m, const Unsigned loc) const {
168   if ( loc == 8 ) {
169     return false ;
170   } else {
171     if ( loc % 2 == 1 ) return edge(loc/2)   . IsOnBoundary(m) ;
172     else                return vertex(loc/2) . IsOnBoundary(m) ;
173   }
174 }
```

Analysis The source fragment implements the public methods `Quad::EqNumber` and `Quad::IsOnBoundary`, see also the comments given for the same methods in the class `Quad` for the $\mathcal{Q}_2$ case. The degrees of freedom are locally enumerated in a counterclockwise order, starting from 0 (a vertex), see Figure 3. The last degree of freedom, identified by 8, is logically associated to the centroid of the quadrilateral element.

The solver code

```
175 void
176 Elliptic_Solver::Solve(pFun f, pFun g,
177                        unsigned const nx, unsigned const ny) {
178   unsigned i, j, k ;

179   // build the mesh
180   mesh . std_tensor_mesh( nx, ny, NULL, NULL, NULL ) ;

181   // allocate memory
182   unsigned neq = mesh.n_vertex() + mesh.n_edge() + mesh.n_poly() ;
183   unsigned nnum = 2*neq + neq * neq ;
184   sol = new double [ nnum ] ;
185   mat = new double * [ neq ] ;
```

```
186  if ( sol == NULL || mat == NULL ) {
187    cerr << "not enought memory" << endl ;
188    exit(0) ;
189  }
190  rhs    = sol + neq ;
191  mat[0] = rhs + neq ;
192  for ( i = 1 ; i < neq ; ++i ) mat[i] = mat[i-1] + neq ;

193  // clean up memory
194  for ( i = 0 ; i < nnum ; ++i ) sol[i] = 0 ;

195  // build the linear system
196  Iterator<Quad> quad(mesh) ;
197  foreach( quad ) {
198    quad -> eval_JJT() ;

199    for ( i = 0 ; i < degree_of_freedom ; ++i ) {
200      if ( quad -> IsOnBoundary(mesh,i) ) continue ;
201      unsigned ig = quad -> EqNumber(mesh,i) ;
202      rhs[ig] += quad -> eval_int_f(i,f) ;

203      for ( j = 0 ; j < degree_of_freedom ; ++j ) {
204        unsigned jg = quad -> EqNumber(mesh,j) ;
205        mat[ig][jg] += quad -> eval_int_grad(i, j) ;
206      }
207    }
208  }

209  // setup boundary conditions
210  Iterator<Vertex> vertex(mesh,1) ;
211  foreach( vertex ) {
212    unsigned ig = vertex -> EqNumber(mesh) ;
213    mat[ig][ig] = 1 ;
214    rhs[ig] = g( vertex -> x(), vertex -> y() ) ;
215  }

216  Iterator<Edge> edge(mesh,1) ;
217  foreach( edge ) {
218    unsigned ig = edge -> EqNumber(mesh) ;
219    mat[ig][ig] = 1 ;
220    rhs[ig] = g( edge -> xm(), edge -> ym() ) ;
221  }

222  // copy rhs to the solution vector
223  for ( i = 0 ; i < neq ; ++i ) sol[i] = rhs[i] ;
```

```
224   // solve the linear system by modified  Gaussian Elimination
225   // without pivoting.
226   cout << "Solving a " << neq << "x" << neq << " linear system"
227        << endl ;
228   for ( i = 0 ; i < neq ; ++i ) {
229     for ( k = 0 ; k < neq ; ++k ) {
230       if ( k != i ) {
231         double bf = mat[k][i]/mat[i][i] ;
232         sol[k] -= bf * sol[i] ;
233         for ( j = i+1 ; j < neq ; ++j )
234           mat[k][j] -= bf * mat[i][j] ;
235       }
236     }
237   }
238   for ( i = 0 ; i < neq ; ++i ) sol[i] /= mat[i][i] ;
239 }
```

Analysis The source fragment implements the method `Solve`. This method is very similar to the one implemented for the triangular case. However, for the sake of clarity, line-by-line comments are repeated. The resolution process can be schematically depicted as follows.

Mesh construction A regular unstructured triangulation of $\mathtt{nx} \cdot \mathtt{ny}$ quadrilaterals over the computational domain $\Omega = (0,1) \times (0,1)$ is built by invoking the `P2MESH` library method `std_tensor_mesh`. About the `NULL` entries and how the boundary markers work, see the comments for the $\mathcal{P}_2$ case.

Memory allocation and initialization The source fragment in lines `175–239` allocates and initializes the arrays used in the construction and resolution of the linear system. In line `182–183` the total memory occupation in terms of double floating point numbers is determined, and then allocated in line `184`. In line `185` the memory required by the matrix row pointers is allocated. In line `190–192` the pointers to the arrays `rhs` and `mat` are initialized and the array values are set up to `0` in line `194`.

Stiffness matrix and right-hand-side vector assembling

This part of the program basically implements in a standard way the construction of the stiffness matrix and the r.h.s. vector. That is, a loop is performed on all the elements of the mesh, and the local contribution to the stiffness matrix and the r.h.s. vector are first evaluated and then assembled into the global arrays. Boundary conditions are taken into account by a direct modification of the final global arrays.

In line `196` the source code instantiates an iterator, called `quad`, to be used for looping on all the quadrilaterals within the mesh. In lines `197–208` the macro `foreach` implements a loop on the mesh cells by using the iterator just introduced. Within the loop, the current quadrilateral is given by the reference returned by the iterator `quad`. In line `198` the method `quad->eval_JJT` returns the values of

$$J_F^{-1}J_F^{-T} \qquad \text{and} \qquad |J_F|$$

which are stored in the matrix `JJF` and in the scalar `detJ`.

The `for` statement in lines `199` and `203` loops on the local degrees of freedom of the current quadrilateral. The integers `ig` and `jg` store the global number corresponding to the local degree of freedom respectively indicated by `i` and `j`. If a degree of freedom is associated to a boundary vertex or edge, the loop is skipped because the corresponding boundary condition is set up in a subsequent part of the program.

In line `202` the local contribution to the r.h.s. vector from the integral

$$\iint\limits_{Q_{ref}} |J_F|\hat{f}(s,t)\hat{v}_i(s,t)\,ds\,dt$$

is estimated and added to the array `rhs`.

In line `205` the components of the local stiffness matrix given by the integral

$$\iint\limits_{Q_{ref}} |J_F|\nabla\hat{v}_i(s,t)^T(J_F^{-1}J_F^{-T})\nabla\hat{v}_j(s,t)\,ds\,dt$$

are evaluated and added to the global stiffness matrix `mat`.

This part is the hard core of the solver.
Line `216` declares and initializes the iterator `quad` to loop over all the mesh quadrilaterals and line `217` uses it within the macro `foreach`. The method `eval_JJT` in line `198` is used to evaluate

$$J_F^{-1}J_F^{-T} \qquad \text{and} \qquad |J_F|$$

on the current quadrilateral at the quadrature points. The values are stored in the static part of the `Quad` class. The source fragment in lines `210–221` modifies the stiffness matrix and the r.h.s vector in order to take into account the boundary conditions on boundary vertices and edges.

The iterator `vertex` is first instantiated in line `210` and initialized to loop on the boundary vertices. Within this loop, implemented in lines `211–215` by using the macro `foreach`, the integer identifier `ig` is set to the degree of freedom associated to the current boundary vertex. In lines `213–214` the stiffness matrix and the r.h.s. vector are modified in correspondence of the `ig`–th row in line `212` in agree with the boundary conditions given by the function `g`.

The boundary conditions for the degree of freedom associated to boundary edges are set up in a similar way. The iterator `edge` is instantiated and initialized to loop on the boundary edges in line `216`, and then utilized within the macro `foreach` in lines `217–221` to implement the loop. In lines `218–220` the identifier `ig` is set to the degree of freedom associated to the current boundary edge, and the `ig`–th row of the stiffness matrix and the r.h.s vector are modified in accord with the boundary conditions given by the function `g`.

Linear system resolution

A modified Gaussian elimination algorithm is implemented in lines `223–238` for the resolution of the linear system. The final solution is stored in the array `sol`.

Saving the computed solution

```
240 void
241 Elliptic_Solver::Save_Mtv(void) {
242   cout << "saving data file..." ;
243   cout . flush() ;
244   ofstream file("q2.mtv") ;
245   file << "$ DATA=CONTCURVE" << endl
246        << "%contstyle=2 meshplot=true topLabel=Q2" << endl ;
247   Iterator<Quad> ip(mesh) ;
248   foreach ( ip ) {
249     for ( unsigned nv = 0 ; nv < 4 ; ++nv ) {
250       Vertex & V = ip -> vertex(nv) ;
251       unsigned i = mesh . local_number(V) ;
252       file << V . x() << " " << V . y() << " " << sol[i] << endl ;
253     }
254     file << endl ;
255   }
```

```
256   file << "$ END" << endl ;
257   file . close() ;
258   cout << "saved" << endl ;
259 }
```

Analysis The final solution stored in the array `sol` is saved on disk in “MTV” format.

The main program

```
260 double Quad::detJ[2][2] ;
261 double Quad::JJT[2][2][2][2] ;
262 double Quad::st[2] = { -0.577350269189626, +0.577350269189626 } ;
```

Analysis The source fragment allocates the static part of the `Quad` class and initialize the vector `st` with the nodal values of the quadrature rule.

```
263 static
264 double
265 f(double const &, double const &)
266 { return -4 ; }

267 static
268 double
269 g(double const & x, double const & y)
270 { return x*x+y*y ; }

271 int
272 main() {
273   Elliptic_Solver es ;
274   es . Solve( f, g, 8, 8) ;
275   es . Save_Mtv() ;
276 }
```

Analysis The driving program defines as static functions the right-hand-side term `f`, see lines `263`–`266`, and the boundary condition term `g`, see lines `267`–`270`. Then, it invokes the methods `Solve` and `Save_Mtv`.

In Figure 4, the final solution computed by the application program is shown.

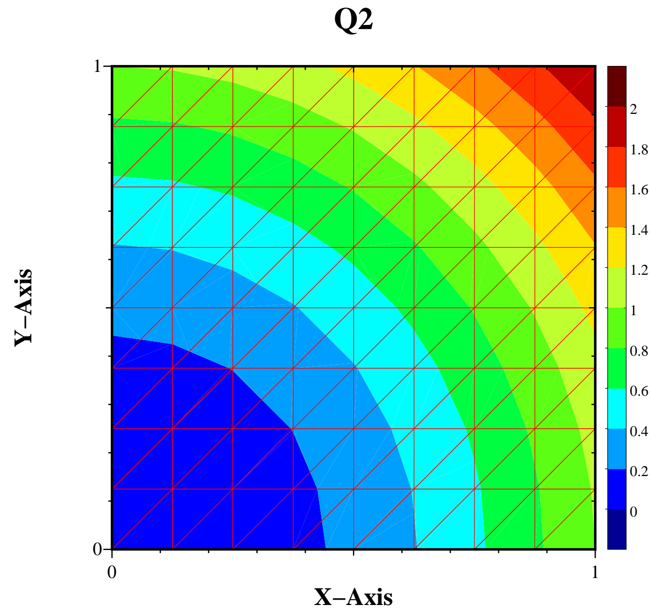

Figure 4: Linear Quadrilateral-based FE Solution of the Poisson problem

6 A FV solver for the compressible Euler equation

In this section two different implementations of a FV solver are presented. The first one implements a cell center method, while the second one a vertex center method. For the sake of simplicity, both schemes are considered only in their simplest version, which is `1`-st order accurate in space. A two stages `2`-nd order accurate Runge-Kutta time-marching scheme advances the solution in time.

The system of compressible Euler equation in 2-D are

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial x} = 0$$

with the conventional notations

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \qquad \mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho\, u \\ \rho\, u^2 + p \\ \rho\, u\, v \\ \rho\, u\, H \end{bmatrix}, \qquad \mathbf{G}(\mathbf{U}) = \begin{bmatrix} \rho\, v \\ \rho\, u\, v \\ \rho\, v^2 + p \\ \rho\, v\, H \end{bmatrix}.$$

and $p = (\gamma - 1)\rho(E - (u^2 + v^2)/2)$ and $H = E + p/\rho$.

A detailed description of solution algorithms for cell center schemes can be found in [4] and for cell vertex schemes in [5] the numerical flux is estimated by using the exact Riemann solver [8].

7 A cell centered finite volume solution

This section describes the program contained in the file `cc_solver.cc` in the directory `examples` of the `P2MESH` distribution package.

Include the library

```
1 # include "p2mesh.hh"
2 # include <math.h>
```

Analysis The statement in line 1 includes the header file `p2mesh.hh`. The pre-processor statement in line 2 includes the header file `math.h` for the standard mathematical function prototypes and definitions.

Declare the project class names

```
3 typedef double Real ;

4 class Vertex ;
5 class Edge ;
6 class Triangle ;
7 class Mesh ;
```

```
8  class Common ;
9  class Solver ;
```

Analysis See the comments given for the FE examples. Notice also that the `typedef` statement in line `3` introduces the alias name `Real` for the floating point built-in type `double`.

Define the class Common

```
10 class Common : public p2_common<Vertex,Edge,Triangle,Mesh,
11                                 3,false,Real> {
12 protected:

13   typedef bool (*PCHECK)   (Real const [4]) ;

14   typedef void (*PFLUX)    (Real [4],
15                             Real [4],
16                             Real const [4]) ;

17   typedef void (*PNUMFLUX) (Real [4],
18                             Real const [4],
19                             Real const [4],
20                             Real const &,
21                             Real const &) ;

22   typedef void (*PCFL)     (Real &,
23                             Real const &,
24                             Real const &,
25                             Real const [4]) ;

26   typedef enum {
27     BC_INTERNAL=0,
28     BC_SUPERSONIC_INLET,
29     BC_SOLID,
30     BC_FREE
31   } BC ;

32 } ;
```

Analysis The piece of source in lines `13–25` contains the `typedefs` of four function pointer types. They are

- PCHECK : function pointer type to functions that perform tests about the numerical and physical consistency of the approximated solution (for example, negative pressures) ;
- PFLUX : function pointer type to functions that compute the physical flux in the cartesian directions ;
- PNUMFLUX : function pointer type to functions that compute the numerical flux in the edge normal direction;
- PCFL : function pointer type to functions that compute the CFL number.

The statements in lines 26–31 define the enum type BC, which is inherited by all the application classes. The BC values are used throughout the application program to discriminate the boundary conditions. Their names are self explanatory.

Define the class Vertex

```
33 class Vertex : public p2_vertex<Common> {} ;
```

Define the class Edge

```
34 class Edge : public p2_edge<Common> {
35   friend class Triangle ;
36   friend class Solver ;
37 private:
38   BC   ibc ;
39   Real num_flux[4] ;
40 public:
41   void InternalNumFlux(PNUMFLUX) ;
42   void BoundaryNumFlux(PNUMFLUX, Real const [4]) ;
43 } ;
```

Analysis The private attributes of the class Edge listed in lines 38–39 are the variable ibc of type BC and a four-element array of type Real, named num_flux. The variable ibc distinguishes whether the edge is an internal or a boundary item, and in the latter case specifies its boundary condition.

The array `num_flux` is assigned the numerical flux along the edge normal direction. Recall that edges have a conventional orientation in the underlying mesh representation provided by `P2MESH` library. The orthogonal direction along which the numerical flux is evaluated is oriented "from left to right" with respect to the orientation of the current edge instance.

The public methods `InternalNumFlux` and `BoundaryNumFlux` respectively evaluate the numerical flux on internal and on boundary edges.

Define the class Triangle

```
44 class Triangle : public p2_poly<Common> {
45   friend class Edge ;
46   friend class Solver ;
47 private:
48   Real hxy, _area, sol[4], sol0[4] ;
49 public:
50   void Init(Real const [4]) ;
51   Real const & area(void) const { return _area ; }
52   void RK_Setsol(void) ;
53   void RK_Update(Real const &, Unsigned const) ;
54 } ;
```

Analysis The private attributes of the class `Triangle` declared in line `39` are

- `hxy` : it is the characteristic size of the triangular cell used in the estimation of the CFL number;
- `_area` : it stores the area of the cell in order to reduce the CPU costs, because the facility provided by `P2MESH` computes the area of the cell each time it is invoked;
- `sol[4]` : it stores the approximate solution at intermediate and final time steps of the Runge-Kutta time marching scheme;
- `sol0[4]` : it stores the initial solution of any Runge-Kutta time step.

The public methods declared in lines `50–53` are

- `Init` : it initializes the class and in particular, the value of the area of the current triangle;
- `area` : it returns the value of the private attribute `_area`; this method overrides the homonymous one inherited from `P2MESH`;
- `RK_Setsol` : it initializes a Runge-Kutta time step;
- `RK_Update` : it performs the solution update in an intermediate Runge-Kutta time step.

Define the class Mesh

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

Define the class Solver

```
56 class Solver : public Common {
57 private:
58   static void mark_edge(Edge & E, Unsigned const & marker) ;

59   Mesh                mesh ;
60   Iterator<Edge>      iedge, bedge ;
61   Iterator<Triangle> triangle ;

62   Unsigned max_iter ;
63   Real     CFL_run, Tend, time, dt ;

64   PCHECK   ok_State ;
65   PNUMFLUX NumFlux ;
66   PFLUX    Flux ;
67   PCFL     CFLxy ;

68   Real inlet_state[4], init_state[4] ;

69 public:
70   Solver(PFLUX, PNUMFLUX, PCHECK, PCFL) ;
71   void SetUp(char const *) ;
72   void SetTimeStep(bool &, Unsigned const);
73   void TimeStep(void) ;
```

```
74   void Save_Mtv(void) ;
75 } ;
```

Analysis The definition of the class Solver is given in lines `56–75`.

As part of the class definition, a private instance of the class `Mesh` is contained, see line `59`.

The statements in lines `60–61` define as private attributes the edge and triangle iterators `iedge`, `bedge` and `triangle`. These iterators are initialized in such a way that

- `iedge` performs loops on the internal edges;
- `bedge` performs loops on the boundary edges;
- `triangle` performs loops on all the mesh triangles.

The variables defined in lines `62–63` are used during the computation of an intermediate Runge-Kutta step:

- `max_iter` : it is the maximum allowable number of time steps;
- `CFL_run` : it is the advancing time step expressed as a fraction of the CFL number;
- `Tend` : it is the final time at which the computation terminates;
- `time` : it is the current time;
- `dt` : it is the current time step.

The statements in lines `64–67` declare the names of the functions which defines the problem. The two four-element sized arrays declared in line `68` are used to store the inflow boundary state (a supersonic inlet) and the initial state of the computation.

The class Solver also contains in lines `70–74` the public methods:

- `Solver` : the constructor links the application program the (externally defined) functions for consistency check, for both the physical and numerical flux calculation, and for the estimation of the CFL number.
- `SetUp` : it reads from an external file the mesh description (in the output format of the mesh generator Triangle), the inlet and the initial state and initialize the mesh representation of `P2MESH` library and some other variables;
- `SetTimeStep` : it computes the new time step `dt`;
- `TimeStep` : it advances the solution of a time step `dt`;
- `Save_Mtv` : it saves on disk the final solution in MTV format.

The methods of the class Edge

```
76 void
77 Edge::InternalNumFlux(PNUMFLUX NumFlux) {
78   Real len = length() ;
79   Real nnx = nx() / len ;
80   Real nny = ny() / len ;
81   NumFlux(num_flux, poly(0).sol, poly(1).sol, nnx, nny) ;
82 }
```

Analysis The statements in lines `78–80` computes the two components of the normalized vector `nnx, nny` orthogonal to the edge. The function `num_flux` called in line `81` evaluates the numerical flux across the edge. The left and right solution states are assigned the solution states at the center of the left and the right triangles (`1`-st order accurate-in-space scheme).

```
83 void
84 Edge::BoundaryNumFlux(PNUMFLUX NumFlux, Real const inlet[4]) {
85   Real len = length() ;
86   Real nnx = nx() / len ;
87   Real nny = ny() / len ;
88   Real rsol[4] ;

89   switch (ibc) {
90   case BC_FREE:
```

```
91      copy(poly(0).sol, poly(0).sol+4, rsol) ;
92    break ;
93    case BC_SUPERSONIC_INLET:
94      copy(inlet, inlet+4, rsol) ;
95    break ;
96    case BC_SOLID:
97      {
98        Real qt = -poly(0).sol[1] * nny + poly(0).sol[2] * nnx ;
99        Real qn = 0 ;
100       rsol[0] = poly(0).sol[0] ;
101       rsol[1] = qn * nnx - qt * nny ;
102       rsol[2] = qn * nny + qt * nnx  ;
103       rsol[3] = poly(0).sol[3] ;
104     }
105   break ;
106   default:
107     cerr << "bad boundary " << (int)ibc << endl ;
108     exit(0) ;
109   }

110   NumFlux(num_flux, poly(0).sol, rsol, nnx, nny) ;
111 }
```

Analysis The statements in lines `85–87` computes the two components of the normalized vector `nnx, nny` orthogonal to the edge. Then, the statements in lines `89–109` set up the right state which corresponds to solution "outside" the computational domain, by taking care of the boundary condition specified by the value of the `BC` variable `ibc`. The function `copy` belongs to the Standard Template Library [3, 6].

Finally, the function `num_flux` called in line `110` evaluates the numerical flux across the edge. The left solution state is the one assigned to the center of the (unique) left triangle adjacent to the boundary edge (`1`-st order accurate-in-space scheme).

The methods of the class Triangle

```
112 inline
113 void
114 Triangle::Init(Real const state[4]) {
115   copy(state, state+4, sol) ;
116   _area = p2_poly<Common>::area() ;
117   hxy = 2*_area/edge(0) . length() ;
118   hxy = min(hxy,2*_area/edge(1) . length()) ;
```

```
119   hxy = min(hxy,2*_area/edge(2) . length()) ;
120 }
```

Analysis Lines 112–120 initialize the current triangle instance. The initial solution state is copied from the input array. The private attribute `_area` is set up by using the inherited function `area`, and the private attribute `hxy` is estimated as the “minimum height” of the triangle.

```
121 inline
122 void
123 Triangle::RK_Setsol(void) {
124   copy(sol, sol+4, sol0) ;
125 }
```

Analysis Lines 121–125 copy the actual state in the buffer variable `sol0`.

```
126 void
127 Triangle::RK_Update(Real const & dt, Unsigned const irk) {
128   // residual
129   Real res[4] ;
130   res[0] = res[1] = res[2] = res[3] = 0 ;
131   for ( Unsigned ie = 0 ; ie < n_edge() ; ++ie ) {
132     Edge & E = edge(ie) ;
133     Real len = ok_oriented(ie) ? E.length() : -E.length() ;
134     res[0] += len * E.num_flux[0] ;
135     res[1] += len * E.num_flux[1] ;
136     res[2] += len * E.num_flux[2] ;
137     res[3] += len * E.num_flux[3] ;
138   }

139   // update
140   static Real crk0[2] = {1, 0.5} ;
141   static Real crk1[2] = {0, 0.5} ;
142   static Real CRKR[2] = {1, 0.5} ;
143   Real crkr = CRKR[irk]*dt/area() ;

144   sol[0] = crk0[irk] * sol0[0] + crk1[irk] * sol[0] - crkr * res[0] ;
145   sol[1] = crk0[irk] * sol0[1] + crk1[irk] * sol[1] - crkr * res[1] ;
146   sol[2] = crk0[irk] * sol0[2] + crk1[irk] * sol[2] - crkr * res[2] ;
147   sol[3] = crk0[irk] * sol0[3] + crk1[irk] * sol[3] - crkr * res[3] ;
148 }
```

Analysis The approximate solution within each triangular cell is advanced in time by applying a two-stage 2-nd order accurate Runge-Kutta scheme also known as the Heun scheme. The approximate solution of the initial ODE problem

$$\begin{cases} \dfrac{du}{dt} = F(u) & , \\ F(0) = F_0 & , \end{cases}$$

satisfies the explicit scheme

$$\begin{aligned} \eta_{k+1} &= u_k + \Delta t F(u_k), \\ u_{k+1} &= u_k + \frac{\Delta t}{2}(F(u_k) + F(\eta_{k+1}). \end{aligned}$$

When applied to the semi-discrete formulation of the cell center FV method

$$|K| \frac{d\mathbf{U}_K}{dt} = - \sum_{e \in \partial K} \boldsymbol{\Phi}_e ,$$

the Runge-Kutta scheme results in the source fragment implemented in the public method `RK_Update`. The local residual

$$- \sum_{e \in \partial K} \boldsymbol{\Phi}_e$$

is estimated in lines `114`–`121`, while lines `144`–`149` advance the solution to the next step.

The solver code

```
149 Solver::Solver(PFLUX    Flux_,
150                PNUMFLUX NumFlux_,
151                PCHECK   ok_State_,
152                PCFL     Cfl_) {
153   Flux     = Flux_ ;
154   NumFlux  = NumFlux_ ;
155   ok_State = ok_State_ ;
156   CFLxy    = Cfl_ ;
157 }
```

Analysis The method assigns the pointers of the functions used in the application program the address of the corresponding (externally defined) functions.

```
158 void
159 Solver::mark_edge(Edge & E, Unsigned const & marker) {
160   switch ( marker ) {
161   case 0 : E.ibc = BC_INTERNAL         ; break ;
162   case 1 : E.ibc = BC_SUPERSONIC_INLET ; break ;
163   case 2 : E.ibc = BC_SOLID            ; break ;
164   case 3 : E.ibc = BC_FREE             ; break ;
165   default:
166     cerr << "mark_edge( E, "<< marker
167          << ") bad boundary condition" << endl ;
168     exit(0) ;
169   }
170 }
```

Analysis The method `mark_edge` processes the markers read in the input file and assigns the variable `ibc` its correct boundary condition. The input file is generated by the mesh generator `triangle`.

In the current example, the input markers correspond to the following situations:

0 internal edge;

1 supersonic inlet edge;

2 solid wall edge;

3 free outlet edge;

```
171 void
172 Solver::SetUp(char const * file) {

173   char file_par[1024] ;
174   strcpy(file_par,file) ;
175   strcat(file_par,".inp") ;
176   ifstream file_input( file_par ) ;

177   if ( ! file_input . good() ) {
```

```
178      cerr << "error in opening file: " << file_par << endl ;
179      exit(0) ;
180    }
181    time = 0 ;
182    file_input
183      >> dt
184      >> Tend
185      >> max_iter
186      >> CFL_run
187      >> inlet_state[0]
188      >> inlet_state[1]
189      >> inlet_state[2]
190      >> inlet_state[3]
191      >> init_state[0]
192      >> init_state[1]
193      >> init_state[2]
194      >> init_state[3] ;
195    cout
196      << "Parameters" << endl
197      << "dt       = " << dt       << endl
198      << "Tend     = " << Tend     << endl
199      << "max_iter = " << max_iter << endl
200      << "CFL_run  = " << CFL_run  << endl
201      << endl
202      << "Input state:"
203      << " r = " << setw(5) << inlet_state[0]
204      << " u = " << setw(5) << inlet_state[1]
205      << " v = " << setw(5) << inlet_state[2]
206      << " E = " << setw(5) << inlet_state[3]
207      << endl
208      << "Initial state:"
209      << " r = " << setw(5) << init_state[0]
210      << " u = " << setw(5) << init_state[1]
211      << " v = " << setw(5) << init_state[2]
212      << " E = " << setw(5) << init_state[3]
213      << endl << endl ;
214    file_input . close() ;
215    // initialize
216    mesh     . read_mesh(file, NULL, mark_edge, NULL, 1) ;
217    bedge    . set_loop(mesh,1) ;
218    iedge    . set_loop(mesh,2) ;
219    triangle . set_loop(mesh) ;
```

```
220   foreach ( triangle ) triangle -> Init(init_state) ;
221 }
```

Analysis The method initializes the run. The string variable `file` stores the basename for the external files containing the initial mesh description, the initial and the inlet flow conditions.

The source fragment in lines `173`–`176` opens the file with the extension "`.inp`". The statements in lines `181`–`194` read all the parameters of the run.

The statements in lines `195`–`213` dump on the standard output the input parameters.

Finally, the statements in lines `216`–`220` initialize the mesh (all the work is done by the function `read_mesh` inherited from the class `p2_mesh`), initialize the private iterators of the solver, and loop on all the triangles to invoke their initialization function.

```
222 void
223 Solver::SetTimeStep(bool & continue_loop, Unsigned const iter) {
224   Real CFL_curr = 0 ;
225   foreach(triangle)
226     CFLxy( CFL_curr, dt, triangle -> hxy, triangle -> sol ) ;

227   Real rapp = min(1.2, CFL_run / CFL_curr) ;
228   dt       *= rapp ;
229   CFL_curr *= rapp ;

230   // chek time step
231   Real new_time = time+dt ;
232   if ( new_time > Tend ) {
233     continue_loop = false ;
234     dt   = Tend - time ;
235     time = Tend ;
236   } else {
237     time = new_time ;
238     continue_loop = continue_loop && iter < max_iter ;
239   }

240   cout
241     << " iter="       << setw(4) << iter
242     << " time (n+1)=" << setw(8) << time
243     << " CFL="        << setw(8) << CFL_curr
244     << " dt="         << setw(8) << dt
245     << endl ;
```

```
246 }
```

Analysis The method computes the new time step. The statements in lines 224–226 compute the current CFL number (and store it in the variable `CFL_curr`). The time step is modified in lines 227–239 as follows:

- if `CFL_curr` is less than `CFL_run`, `dt` is increased of at most the 20% of its current value;
- if `CFL_curr` is greater than `CFL_run`, `dt` is reduced of the ratio `CFL_run / CFL_curr`;

In lines 231–239 the program checks if the run terminates at the current iteration. The boolean variable `continue_loop` is set up consequently. Finally, the statements in lines 240–245 print on standard output some useful information.

```
247 void
248 Solver::TimeStep(void) {
249   foreach(triangle) triangle -> RK_Setsol() ;
250   for( Unsigned irk = 0 ; irk < 2 ; ++irk ) {
251     foreach(iedge) iedge -> InternalNumFlux(NumFlux) ;
252     foreach(bedge) bedge -> BoundaryNumFlux(NumFlux,inlet_state) ;
253     foreach(triangle) {
254       triangle -> RK_Update(dt,irk) ;
255       if ( !ok_State(triangle -> sol) ) {
256         cerr << "POSITIVITY_CHECK: negative pressure found" ;
257         exit(0) ;
258       }
259     }
260   }
261 }
```

Analysis This piece of code advances the solution to the next step. In line 249 the Runge-Kutta scheme is initialized. Lines 251–252 compute the numerical flux of internal and boundary edges. Line 254 performs the `irk`-th stage of the Runge-Kutta stepping scheme.

Saving the computed solution

```
262 void
263 Solver::Save_Mtv(void) {
264   ofstream file("cc.mtv") ;
265   if ( ! file . good() ) {
266     cerr << "Cannot open for write file: ``cc.mtv''" << endl ;
267     exit(0) ;
268   }
269   file << "$ DATA=CONTCURVE\n%contstyle=2 topLabel=mass"
270        << endl ;
271   foreach ( triangle ) {
272     Real fun = triangle -> sol[0] ;
273     file << triangle->x(0) << " " << triangle->y(0) << " " << fun
274          << endl
275          << triangle->x(1) << " " << triangle->y(1) << " " << fun
276          << endl
277          << triangle->x(2) << " " << triangle->y(2) << " " << fun
278          << endl << endl ;
279   }
280   file << "$ END" << endl ;
281   file . close() ;
282 }
```

Analysis The method saves the solution in MTV format.

The main program

```
283 # include "eu.hh"
284 int
285 main() {
286   Solver solver(Euler::Flux,
287                 Euler::Godunov,
288                 Euler::ok_State,
289                 Euler::CFL) ;
290   // set input data
291   solver . SetUp("ramp") ;
```

```
292   // advancing loop
293   bool continue_loop = true  ;
294   for ( unsigned iter = 1 ; continue_loop ; ++iter ) {
295     solver.SetTimeStep(continue_loop, iter) ; // variable time step dt
296     solver.TimeStep() ; // update one time step
297   } ;

298   solver . Save_Mtv() ;
299   cout << "End of Program" << endl ;

300 }
```

Analysis Line 283 includes the header file `eu.hh`. This file contains the implementation of the physical flux of the Euler equations and the Godunov and Lax-Friedrics numerical fluxes. The statements in lines 286–289 instantiate and initialize an object of the class `Solver`. Notice that the constructor takes in input the addresses of the functions defined in the class `Euler`.

Figure 5 shows the final solution computed by this application program.

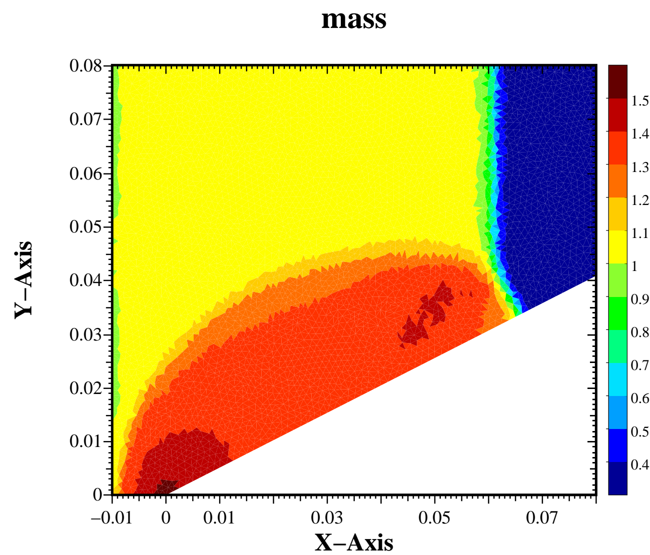

Figure 5: Double Mach reflection solution produced by the cell center FV Euler solver

8 A vertex centered finite volume solution

This section describes the program contained in the file `cc_solver.cc` in the directory `examples` of the `P2MESH` distribution package.

Include the library

```
1 # include "p2mesh.hh"
2 # include <math.h>
```

Analysis The source fragment includes the library header file `p2mesh.hh` and the standard C++ header file `math.h` for the mathematical function prototypes and definitions.

Declare user-defined class names

```
3 typedef double Real ;

4 class Vertex ;
5 class Edge ;
6 class Triangle ;
7 class Mesh ;
8 class Common ;
9 class Solver ;
```

Analysis See the comments given for the earlier example.

Define the class Common

```
10 class Common : public p2_common<Vertex,Edge,Triangle,Mesh,
11                                 3,true,Real> {
12 protected:

13   typedef bool (*PCHECK)   (Real const [4]) ;

14   typedef void (*PFLUX)    (Real [4],
15                             Real [4],
16                             Real const [4]) ;
```

```
17   typedef void (*PNUMFLUX) (Real [4],
18                              Real const [4],
19                              Real const [4],
20                              Real const &,
21                              Real const &) ;

22   typedef void (*PCFL)     (Real &,
23                              Real const &,
24                              Real const &,
25                              Real const [4]) ;

26   typedef enum {
27     BC_INTERNAL=0,
28     BC_SUPERSONIC_INLET,
29     BC_SOLID,
30     BC_FREE
31   } BC ;

32 } ;
```

Analysis The actual implementation makes usage of the topological lists of the vertices, edges and triangles incident on a given vertex. `P2MESH` automatically initializes such lists when we set `LIST = true` in the template argument list of `p2_common` in lines `10–11`.

Define the class Vertex

```
33 class Vertex : public p2_vertex<Common> {
34 private:
35   friend class Edge ;
36   friend class Solver ;
37   Real _area, hxy, sol[4], sol0[4] ;
38 public:
39   void Init(Real const[4]) ;
40   Real const & area(void) const { return _area ; }
41   void RK_Setsol(void) ;
42   void RK_Update(Real const &, Unsigned const) ;
43 } ;
```

Analysis The class Vertex contains in line `37` the following private attributes

- `hxy` : it is the characteristic size of the vertex control volume on the dual mesh used to estimate the CFL number;
- `_area` : it stores the area of the control volume to reduce the CPU costs of area calculations;
- `sol[4]` : it stores the approximate solution at any intermediate and final time step of the Runge-Kutta time marching scheme;
- `sol0[4]` : (work array) it stores the initial solution of any Runge-Kutta time step;

and the following public methods

- `Init` : it initializes the class; in particular, compute the the value of the area of the control volume of the current vertex instance;
- `area` : it returns the value of the private attribute `_area`;
- `RK_Setsol` : it initializes a Runge-Kutta time step;
- `RK_Update` : it performs the solution update in an intermediate Runge-Kutta time step.

Notice that in a cell vertex scheme the application unknowns are logically associated to the vertices, which naturally implies this very rich definition of the vertex data structure. Compare this definition with the one given for the previous cell center implementation.

Define the class Edge

```
44 class Edge : public p2_edge<Common> {
45 private:
46   friend class Vertex ;
47   friend class Solver ;
48   BC   ibc ;
49   Real num_flux[2][4], nx[2], ny[2], len[2] ;
50 public:
51   void Init(void) ;
```

```
52   void InternalNumFlux(PNUMFLUX) ;
53   void BoundaryNumFlux(PNUMFLUX, Real const [4]) ;
54 } ;
```

Analysis The private attribute of the class `Edge` are the variable `ibc` of type `BC`, and four arrays of type `Real`, `num_flux`, `nx`, `ny`, and `len`:

- `ibc` : it distinguishes the nature of the current edge instance, (internal or on the boundary), and eventually specifies a boundary condition;
- `nx` : it stores the first component of the two non-normalized vectors orthogonal to the medians from the left (`0`) and right (`1`) adjacent triangles;
- `ny` : it stores the second component of the two non-normalized vectors orthogonal to the medians from the left (`0`) and right (`1`) adjacent triangles;
- `len` : it stores the two length of the segment joining the edge midpoint with the centroid of the left (`0`) and right (`1`) adjacent triangles; notice that $\texttt{len[i]} = \texttt{nx[i]}^2 + \texttt{ny[i]}^2$, where `i=0`,`1`;
- `num_flux` : it contains the numerical flux across the two segments joining the edge midpoint with the centroid of the left (`0`) and right (`1`) adjacent triangles;

Notice that whenever the current edge is located on the boundary of the computational domain, half of the edge coincides with a portion of the control volume boundary. In this case, (`nx[1]`, `ny[1]`) defines a vector whose direction is outward oriented and orthogonal to the edge and whose length `len` is half of the edge length.

The public methods of the class `Edge` are

- `Init` : it computes `nx`, `ny`, and `len`;
- `InternalNumFlux` : it computes the numerical flux of internal edges;
- `BoundaryNumFlux` : it computes the numerical flux of boundary edges.

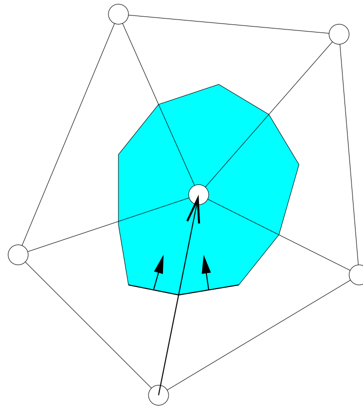

Figure 6:

Define the class Triangle

```
55 class Triangle : public p2_poly<Common> {} ;
```

Analysis The body of the class `Triangle` is empty as a consequence of the cell vertex nature of the numerical algorithm. Compare this definition with the one given for the cell center scheme implementation.

Define the class Mesh

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

Define the main solver class

```
57 class Solver : public Common {
58 private:
59   static void mark_edge(Edge &, Unsigned const &) ;

60   Mesh                mesh ;
61   Iterator<Vertex>    vertex ;
62   Iterator<Edge>      edge, iedge, bedge ;
```

```
63   Iterator<Triangle> triangle ;

64   Unsigned max_iter ;
65   Real     CFL_run, Tend, time, dt ;

66   PCHECK   ok_State ;
67   PNUMFLUX NumFlux ;
68   PFLUX    Flux ;
69   PCFL     CFLxy ;

70   Real inlet_state[4], init_state[4] ;

71 public:
72   Solver(PFLUX, PNUMFLUX, PCHECK, PCFL);
73   void SetUp(char const *) ;
74   void SetTimeStep(bool &, Unsigned const);
75   void TimeStep(void) ;
76   void Save_Mtv(void) ;
77 } ;
```

Analysis The definition of the class Solver is given in lines `57–77`.

As part of the class definition, a private instance of the class `Mesh` is contained, see line `60`.

The private attributes in lines `61–63` are the iterators `vertex`, `iedge`, `bedge` and `triangle`. The iterators are initialized in such a way that

- `vertex` : it performs loops on all the vertices;
- `edge` : it performs loops on all the edges;
- `iedge` : it performs loops on the internal edges;
- `bedge` : it performs loops on the boundary edges;
- `triangle` : it performs loops on all the triangles.

The variables defined in lines `64–65` are used during the computation of an intermediate Runge-Kutta step:

- max_iter : it is the maximum allowable number of time steps;
- CFL_run : it is the time step size expressed as a fraction of the CFL number;
- Tend : it is the final time at which the computation terminates;
- time : it is the current time;
- dt : it is the current time step.

The statements in lines 66–69 declare the names of the functions that define the problem. The two arrays in line 70 store the inflow boundary state (a supersonic inlet) and the initial state of the computation.

The class Solver contains in lines 72–76 the following public methods:

- Solver : the constructor links the application program the (externally defined) functions for consistency check, for both physical and numerical flux calculation, and for the estimation of the CFL number.
- SetUp : it reads from an external file the mesh description (in the output format of the mesh generator Triangle), the inlet and the initial state and initialize the mesh representation of P2MESH library and some other variables;
- SetTimeStep : it computes the new time step dt;
- TimeStep : it advances the solution of a time step dt;
- Save_Mtv : it saves on disk the final solution in MTV format.

The methods of the class Vertex

```
78 inline
79 void
80 Vertex::Init(Real const state[4]) {
81   copy(state, state+4, sol) ;
82   unsigned i ;
83   _area = 0 ;
```

```
84    for ( i = 0 ; i < n_poly() ; ++i )
85      _area += poly(i).area() ;
86    _area /= 3 ;

87    hxy = edge(0) . length() ;
88    for ( i = 1 ; i < n_edge() ; ++i )
89      hxy = min(hxy, edge(i) . length() ) ;

90  }
```

Analysis The method initializes the class `Vertex`. The area of the control volume associated to the current vertex is evaluated and assigned to the variable `_area`. The method also initializes the value of `hxy`, which is given by the minimum length of the internal edges of the control volume, needed in the estimation of the CFL.

```
91  inline
92  void
93  Vertex::RK_Setsol(void) {
94    copy(sol, sol+4, sol0) ;
95  }
```

Analysis See the comment given in the cell center example.

```
96  void
97  Vertex::RK_Update(Real const & dt, Unsigned const irk) {

98    // compute residual
99    Real res[4] ;
100   res[0] = res[1] = res[2] = res[3] = 0 ;
101   for ( Unsigned ie = 0 ; ie < n_edge() ; ++ie ) {
102     Edge & E = edge(ie) ;
103     bool ok_dir = this == &E.vertex(0) ;
104     Real len = ok_dir ? E.len[0] : -E.len[0] ;
105     res[0] += len * E.num_flux[0][0] ;
106     res[1] += len * E.num_flux[0][1] ;
107     res[2] += len * E.num_flux[0][2] ;
108     res[3] += len * E.num_flux[0][3] ;

109     len = (ok_dir || E.ibc != BC_INTERNAL) ? E.len[1] : -E.len[1] ;
110     res[0] += len * E.num_flux[1][0] ;
```

```
111      res[1] += len * E.num_flux[1][1] ;
112      res[2] += len * E.num_flux[1][2] ;
113      res[3] += len * E.num_flux[1][3] ;
114    }

115    // update
116    static Real crk0[2] = {1, 0.5} ;
117    static Real crk1[2] = {0, 0.5} ;
118    static Real CRKR[2] = {1, 0.5} ;
119    Real crkr = CRKR[irk]*dt/area() ;

120    sol[0] = crk0[irk] * sol0[0] + crk1[irk] * sol[0] - crkr * res[0] ;
121    sol[1] = crk0[irk] * sol0[1] + crk1[irk] * sol[1] - crkr * res[1] ;
122    sol[2] = crk0[irk] * sol0[2] + crk1[irk] * sol[2] - crkr * res[2] ;
123    sol[3] = crk0[irk] * sol0[3] + crk1[irk] * sol[3] - crkr * res[3] ;

124  }
```

Analysis The method computes in lines 99–114 the residual

$$-\sum_{e\in\partial K} \boldsymbol{\Phi}_e$$

associated to the control volume of the vertex, which is then used in lines 120–123 to advance the solution to the next Runge-Kutta stage.

The methods of the class Edge

```
125  void
126  Edge::Init(void) {
127    nx[0] = poly(0).yc() - ym() ;
128    ny[0] = xm() - poly(0).xc() ;

129    if ( ok_poly(1) ) {
130      nx[1] = ym() - poly(1).yc() ;
131      ny[1] = poly(1).xc() - xm() ;
132    } else {
133      nx[1] = p2_edge<Common>::nx()/2 ;
134      ny[1] = p2_edge<Common>::ny()/2 ;
135    }

136    for ( Unsigned i = 0 ; i < 2 ; ++i ) {
```

```
137      len[i] = sqrt( nx[i]*nx[i] + ny[i]*ny[i] ) ;
138      nx[i] /= len[i] ;
139      ny[i] /= len[i] ;
140    }
141  }
```

Analysis The method computes (`nx[0]`,`ny[0]`) and (`nx[1]`,`ny[1]`). These two vectors are orthogonal to the segment which joins the edge midpoint and the two centroids of the left and right triangles. It also computes the length of the segments, since the two orthogonal vectors are non-normalized. The test on line `129` discriminates the boundary edges.

```
142  void
143  Edge::InternalNumFlux(PNUMFLUX NumFlux) {
144    Vertex & VA = vertex(0) ;
145    Vertex & VB = vertex(1) ;
146    NumFlux(num_flux[0], VA.sol, VB.sol, nx[0], ny[0]) ;
147    NumFlux(num_flux[1], VA.sol, VB.sol, nx[1], ny[1]) ;
148  }
```

Analysis The statements in lines `146–147` invoke the function `NumFlux`, which computes the numerical fluxes across the left and right portion of the control volume boundary associated to the current edge instance.

```
149  void
150  Edge::BoundaryNumFlux(PNUMFLUX NumFlux, Real const inlet[4]) {
151    Vertex & VA = vertex(0) ;
152    Vertex & VB = vertex(1) ;

153    NumFlux(num_flux[0], VA.sol, VB.sol, nx[0], ny[0]) ;

154    Real lsol[4], rsol[4] ;
155    lsol[0] = 0.5*(VA.sol[0] + VB.sol[0]) ;
156    lsol[1] = 0.5*(VA.sol[1] + VB.sol[1]) ;
157    lsol[2] = 0.5*(VA.sol[2] + VB.sol[2]) ;
158    lsol[3] = 0.5*(VA.sol[3] + VB.sol[3]) ;

159    switch (ibc) {
160    case BC_FREE:
161      copy(lsol, lsol+4, rsol) ;
```

```
162    break ;
163    case BC_SUPERSONIC_INLET:
164      copy(inlet, inlet+4, rsol) ;
165    break ;
166    case BC_SOLID:
167      {
168        Real qt = -lsol[1] * ny[1] + lsol[2] * nx[1] ;
169        Real qn = 0 ;
170        rsol[0] = lsol[0] ;
171        rsol[1] = qn * nx[1] - qt * ny[1] ;
172        rsol[2] = qn * ny[1] + qt * nx[1]  ;
173        rsol[3] = lsol[3] ;
174      }
175    break ;
176    default:
177      cerr << "bad boundary " << (int)ibc << endl ;
178      exit(0) ;
179    }

180    NumFlux(num_flux[1], lsol, rsol, nx[1], ny[1]) ;
181  }
```

Analysis As for the cell center scheme, the numerical flux in the case of a boundary edges is given by a different method, which takes into account the boundary conditions.

The solver code

```
182  Solver::Solver(PFLUX    Flux_,
183                 PNUMFLUX NumFlux_,
184                 PCHECK   ok_State_,
185                 PCFL     Cfl_) {
186    Flux     = Flux_ ;
187    NumFlux  = NumFlux_ ;
188    ok_State = ok_State_ ;
189    CFLxy    = Cfl_ ;
190  }
```

Analysis See the comments given for the cell center case.

```
191  void Solver::mark_edge(Edge & E, Unsigned const & marker) {
192    switch ( marker ) {
```

```
193   case 0 : E.ibc = BC_INTERNAL          ; break ;
194   case 1 : E.ibc = BC_SUPERSONIC_INLET ; break ;
195   case 2 : E.ibc = BC_SOLID             ; break ;
196   case 3 : E.ibc = BC_FREE              ; break ;
197   default:
198     cerr << "mark_edge( E, " << marker
199          << ") bad boundary condition"  << endl ;
200     exit(0) ;
201   }
202 }
```

Analysis See the comments given for the cell center case.

```
203 void Solver::SetUp(char const * file) {

204   char file_par[1024] ;
205   strcpy(file_par,file) ;
206   strcat(file_par,".inp") ;

207   ifstream file_input( file_par ) ;

208   if ( ! file_input . good() ) {
209     cerr << "error in opening file: " << file_par << endl ;
210     exit(0) ;
211   }

212   time = 0 ;
213   file_input
214     >> dt
215     >> Tend
216     >> max_iter
217     >> CFL_run
218     >> inlet_state[0]
219     >> inlet_state[1]
220     >> inlet_state[2]
221     >> inlet_state[3]
222     >> init_state[0]
223     >> init_state[1]
224     >> init_state[2]
225     >> init_state[3] ;

226   cout
227     << "Parameters" << endl
228     << "dt       = " << dt       << endl
```

```
229     << "Tend     = " << Tend     << endl
230     << "max_iter = " << max_iter << endl
231     << "CFL_run  = " << CFL_run  << endl
232     << endl
233     << "Input state:"
234     << " r = " << setw(5) << inlet_state[0]
235     << " u = " << setw(5) << inlet_state[1]
236     << " v = " << setw(5) << inlet_state[2]
237     << " E = " << setw(5) << inlet_state[3]
238     << endl
239     << "Initial state:"
240     << " r = " << setw(5) << init_state[0]
241     << " u = " << setw(5) << init_state[1]
242     << " v = " << setw(5) << init_state[2]
243     << " E = " << setw(5) << init_state[3]
244     << endl << endl ;

245   file_input . close() ;

246   // initialize
247   mesh     . read_mesh(file, NULL, mark_edge, NULL, 1) ;
248   vertex   . set_loop(mesh) ;
249   edge     . set_loop(mesh) ;
250   bedge    . set_loop(mesh,1) ;
251   iedge    . set_loop(mesh,2) ;
252   triangle . set_loop(mesh) ;

253   foreach ( vertex ) vertex -> Init(init_state) ;
254   foreach ( edge   ) edge   -> Init() ;

255 }
```

Analysis This source fragment is similar to the corresponding one of the cell center case. The main difference is in the initialization phase performed on the vertices and not on the triangles.

```
256 void
257 Solver::SetTimeStep(bool & continue_loop, Unsigned const iter) {
258   Real CFL_curr = 0 ;
259   foreach(vertex)
260     CFLxy( CFL_curr, dt, vertex -> hxy, vertex -> sol ) ;

261   Real rapp = min(1.2, CFL_run / CFL_curr) ;
```

```
262   dt       *= rapp ;
263   CFL_curr *= rapp ;

264   // chek time step
265   Real new_time = time+dt ;
266   if ( new_time > Tend ) {
267     continue_loop = false ;
268     dt   = Tend - time ;
269     time = Tend ;
270   } else {
271     time = new_time ;
272     continue_loop = continue_loop && iter < max_iter ;
273   }

274   cout
275     << " iter="       << setw(4) << iter
276     << " time (n+1)=" << setw(8) << time
277     << " CFL="        << setw(8) << CFL_curr
278     << " dt="         << setw(8) << dt
279     << endl ;

280 }
```

Analysis The method computes the new time step by looping on all the vertices. Since this is the only difference with respect to the cell center case, see the comments given therein.

```
281 void
282 Solver::TimeStep(void) {
283   foreach(vertex) vertex -> RK_Setsol() ;
284   for( Unsigned irk = 0 ; irk < 2 ; ++irk ) {
285     foreach(iedge) iedge -> InternalNumFlux(NumFlux) ;
286     foreach(bedge) bedge -> BoundaryNumFlux(NumFlux,inlet_state) ;
287     foreach(vertex) {
288       vertex -> RK_Update(dt,irk) ;
289       if ( !ok_State(vertex -> sol) ) {
290         cerr << "POSITIVITY_CHECK: negative pressure found" ;
291         exit(0) ;
292       }
293     }
294   }
295 }
```

Analysis This source fragment is almost identical to the one given for the cell center case. The only difference is in the loop which is performed on vertices instead of on triangles.

Saving the computed solution

```
296 void
297 Solver::Save_Mtv(void) {
298   ofstream file("cv.mtv") ;
299   if ( ! file . good() ) {
300     cerr << "Cannot open for write file: ‘‘cv.mtv’’" << endl ;
301     exit(0) ;
302   }
303   file << "$ DATA=CONTCURVE\n%contstyle=2 topLabel=mass"
304        << endl ;
305   foreach ( triangle ) {
306     for ( Unsigned nv = 0 ; nv < triangle -> n_vertex() ; ++nv ) {
307       Vertex & V = triangle -> vertex(nv) ;
308       file << V.x() << " " << V.y() << " " << V.sol[0] << endl ;
309     }
310     file << endl ;
311   }
312   file << "$ END" << endl ;
313   file . close() ;
314 }
```

Analysis The method saves the computed solution in a MTV format file.

The main program

```
315 # include "eu.hh"
316 int
317 main() {
318   Solver solver(Euler::Flux,
319                 Euler::Godunov,
320                 Euler::ok_State,
321                 Euler::CFL) ;
322   solver . SetUp("ramp") ;
```

```
323    bool continue_loop = true  ;
324    for ( unsigned iter = 1 ; continue_loop ; ++iter ) {
325      solver.SetTimeStep(continue_loop, iter) ; // variable time step dt
326      solver.TimeStep() ; // update one time step
327    } ;

328    solver . Save_Mtv() ;
329    cout << "End of Program" << endl ;

330  }
```

Analysis In line `315` the header file `eu.hh` is included. This file contains the implementations of the physical flux for the Euler equations and of the Godunov and Lax-Friedrics numerical fluxes. The statements in lines `318–321` instantiate and initialize an object of the class `Solver`. Notice that the constructor takes in input the addresses of the functions defined in the class `Euler`.

Figure 7 shows the final solution computed by this application program.

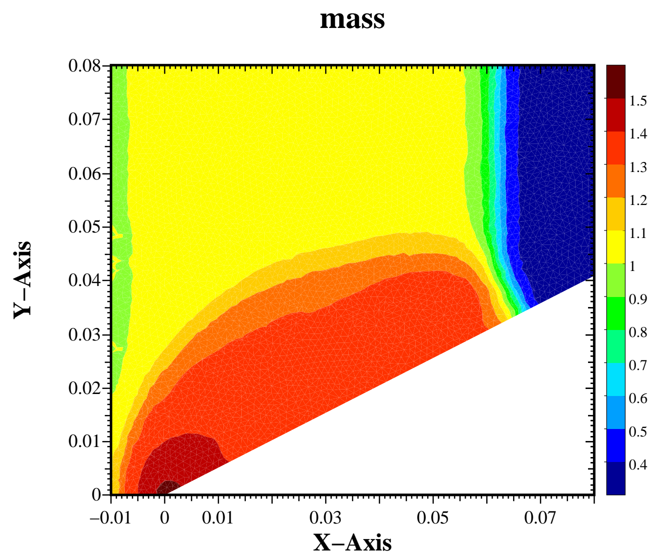

Figure 7: Solution produced by the double Mach reflection vertex-center FV solver

A The file "eu.hh"

```
class Euler {
public:
  typedef double Real ;

private:
  // private internal stuff

public:
  static void CFL(Real &,
                  Real const &, Real const &, Real const &,
                  Real const [4]) ;
  static bool ok_State(Real const [4]) ;

  static void Flux(Real [4], Real [4], Real const [4]) ;

  static void LF(Real [4], Real const [4], Real const [4],
                 Real const & , Real const & ) ;

  static void Godunov(Real [4], Real const [4], Real const [4],
                      Real const &, Real const &) ;

} ;
```

B The file "eu.cc"

```
# include "eu.hh"

# include <iostream>
# include <math.h>

typedef Euler::Real Real ;

inline Real abs(Real const & a)
{ return a > 0 ? a : -a ; }

inline Real max(Real const & a, Real const & b)
{ return a > b ? a : b ; }
```

```
inline Real min(Real const & a, Real const & b)
{ return a < b ? a : b ; }

static Real const GAMMA = 1.4 ;
static Real const G1 = (GAMMA - 1) / (2 * GAMMA) ;
static Real const G2 = (GAMMA + 1) / (2 * GAMMA) ;
static Real const G3 = 2 * GAMMA / (GAMMA - 1) ;
static Real const G4 = 2 / (GAMMA - 1) ;
static Real const G5 = 2 / (GAMMA + 1) ;
static Real const G6 = (GAMMA - 1) / (GAMMA + 1) ;
static Real const G7 = (GAMMA - 1) / 2 ;
static Real const G8 = 1 / GAMMA ;
static Real const G9 = GAMMA - 1 ;

Real Euler::ec(Real const val[4]) {
  return 0.5 * ( val[1]*val[1] + val[2]*val[2] ) / r(val) ;
}

Real Euler::P(Real const val[4]) {
  Real press = G9 * ( E(val) - ec(val) ) ;
  if ( press <= 0 ) {
    cerr
      << "Euler::P("
      << val[0] << ","
      << val[1] << ","
      << val[2] << ","
      << val[3] << ") found bad pressure p = "
      << press << endl ;
    exit(0) ;
  }
  return press ;
}

Real Euler::C(Real const val[4]) {
  Real C2 = GAMMA * P(val) / r(val) ;
  if ( C2 <= 0 ) {
    cerr
      << "Euler::C("
      << val[0] << ","
      << val[1] << ","
      << val[2] << ","
      << val[3] << ") found bad speed C^2 = "
      << C2 << endl ;
    exit(0) ;
  }
```

```
  return sqrt(C2) ;
}

bool Euler::ok_State(Real const val[4]) {
  Real press = G9 * ( E(val) - ec(val) ) ;
  return press > 0 && r(val) > 0 ;
}

void Euler::CFL(Real       & CFL,
                Real const & dt,
                Real const & h,
                Real const   val[4] ) {
  Real c = C(val) ;
  Real u = U(val) ; if ( u < 0 ) u = -u ;
  Real v = V(val) ; if ( v < 0 ) v = -v ;

  CFL = max( dt*(max(u,v)+c)/h, CFL ) ;
}

void Euler::Flux(Real fx[4], Real fy[4], Real const val[4]) {

  Real u = U(val) ;
  Real v = V(val) ;

  for ( unsigned i = 0 ; i < 4 ; ++i ) {
    fx[i] = u * val[i] ;
    fy[i] = v * val[i] ;
  }

  Real press = P(val) ;

  fx[1] += press ;
  fx[3] += press * u ;

  fy[2] += press ;
  fy[3] += press * v ;

}

void Euler::LF(Real         nflux[4],
               Real const   lsol[4],
               Real const   rsol[4],
               Real const & nx,
               Real const & ny) {

  Real vl = nx * U(lsol) + ny * V(lsol) ;
```

```
  Real vr = nx * U(rsol) + ny * V(rsol) ;
  Real artvisc = max( abs(vl) + C(lsol), abs(vr) + C(rsol)) ;

  Real lfx[4], lfy[4], rfx[4], rfy[4] ;

  Flux(lfx, lfy, lsol) ;
  Flux(rfx, rfy, rsol) ;

  for ( unsigned i = 0 ; i < 4 ; ++i )
    nflux[i] = 0.5 * ( nx * (lfx[i]+rfx[i]) + ny * (lfy[i]+rfy[i])
                       - artvisc * ( rsol[i] - lsol[i] ) ) ;
}

void Euler::StateEval(State      & s,
                      Real const   S[4],
                      Real const & nx,
                      Real const & ny ) {

  s.r = r(S) ;
  s.u = vn(S,nx,ny) ;
  s.p = P(S) ;
  s.c = GAMMA * s.p / r(S) ;
  if ( s.c <= 0 ) {
    cerr << "error in StateEval negative sound speed" << endl ;
    exit(0) ;
  }
  s.c = sqrt( s.c ) ;

}

//----------------------------------------------------------------

void Euler::Godunov(Real         nflux[4],
                    Real const   lsol[4],
                    Real const   rsol[4],
                    Real const & nx,
                    Real const & ny) {

  if ( !ok_State(lsol) ) {
    cerr << "Euler::Godunov bad left state" << endl ;
    exit(0) ;
  }

  if ( !ok_State(rsol) ) {
    cerr << "Euler::Godunov bad right state" << endl ;
    exit(0) ;
```

```
  }

  Real const SS = 0 ;
  State sl, sr, sm ;
  StateEval( sl, lsol, nx, ny ) ;
  StateEval( sr, rsol, nx, ny ) ;

  Real PM, UM ;
  // LOCAL RIEMANN PROBLEM RP(I,I+1) IS SOLVED EXACTLY
  Riemann(PM, UM, sl, sr) ;
  // SOLUTION IS SAMPLED AT S=X/T=0 ALONG T-AXIS
  Sample(PM, UM, SS, sl, sr, sm) ;

  Real qn = sm.u ;
  Real qt = qn > 0 ? vt(lsol,nx,ny) : vt(rsol,nx,ny) ;
  Real u  = qn * nx - qt * ny ;
  Real v  = qn * ny + qt * nx ;

  nflux[0] = qn * sm.r ;
  nflux[1] = qn * sm.r * u + sm.p * nx ;
  nflux[2] = qn * sm.r * v + sm.p * ny ;
  nflux[3] = qn * (sm.p+sm.p/G9 + 0.5*sm.r*(u*u+v*v)) ;
}

//---------------------------------------------------------------

void Euler::Riemann(Real        & P,
                    Real        & U,
                    State const & sl,
                    State const & sr) {
  // COMPUTE PRESSURE PM AND PARTICLE VELOCITY UM IN THE MIDDLE
  // PM IS FOUND ITERATIVELY BY A NEWTON-RAPHSON METHOD.

  // COMPUTE GUESS VALUE FROM PVRS RIEMANN SOLVER
  Real PPV = 0.5*(sl.p+sr.p)
           - 0.125*(sr.u-sl.u)*(sl.r+sr.r)*(sl.c+sr.c) ;
  Real PMIN = min(sl.p, sr.p) ;
  Real PMAX = max(sl.p, sr.p) ;
  Real QRAT = PMAX / PMIN  ;

  if ( QRAT <= 2.0 && (PMIN <= PPV && PPV <= PMAX) ) {
    // USE PVRS SOLUTION AS GUESS
    P = PPV ;
  } else {
    if (PPV < PMIN) { // USE TWO-RAREFACTION SOLUTION
      Real PNU = sl.c + sr.c - G7 * (sr.u - sl.u) ;
```

```
      Real PDE = sl.c / pow(sl.p, G1 ) + sr.c / pow( sr.p, G1 ) ;
      P = pow(PNU / PDE, G3 ) ;
    } else { // USE TWO-SHOCK APPROXIMATION WITH PPV AS ESTIMATE
      Real GEL = sqrt( (G5 / sl.r) / (G6 * sl.p + PPV) ) ;
      Real GER = sqrt( (G5 / sr.r) / (G6 * sr.p + PPV) ) ;
      P = (GEL*sl.p + GER*sr.p - (sr.u-sl.u) ) / (GEL + GER) ;
    }
   }

  Real const TOL = 1e-6 ;
  Real FL, FR, FLD, FRD ;
  Real P0 = P ;
  Real DU = sr.u - sl.u ;
  for ( unsigned k = 0 ; k < 50 ; ++k ) {
    Prefun(FL, FLD, P, sl) ;
    Prefun(FR, FRD, P, sr) ;
    P -= (FL + FR + DU) / (FLD+FRD) ;
    if ( abs( (P - P0) / (P + P0) ) <= 0.5 * TOL ) goto fine ;
    P0 = P > 0 ? P : TOL ;
  }
  cout << "Euler::Riemann(...)"
       << "DIVERGENCE IN NEWTON-RAPHSON ITERATION" << endl ;

fine:
  // COMPUTE U
  U = 0.5 * (sl.u + sr.u + FR - FL) ;
}

void Euler::Prefun(Real        & F,
                   Real        & FD,
                   Real  const & P,
                   State const & s) {
  if (P <= s.p) { // RAREFACTION WAVE
    Real PRAT = P / s.p ;
    F  = G4 * s.c * (pow(PRAT,G1) - 1) ;
    FD = (1.0 / (s.r * s.c) ) * pow(PRAT, -G2 ) ;
  } else { // SHOCK WAVE
    Real AK = G5 / s.r ;
    Real BK = G6 * s.p ;
    Real QRT = sqrt(AK / (BK + P) ) ;
    F  = (P - s.p) * QRT ;
    FD = (1 - 0.5 * (P - s.p) / (BK + P) ) * QRT ;
  }
}

void Euler::Sample(Real  const & PM,
```

```
                 Real  const & UM,
                 Real  const & S,
                 State const & sl,
                 State const & sr,
                 State       & sm ) {
if ( S <= UM ) {
  // SAMPLE POINT IS TO THE LEFT OF THE CONTACT
  if ( PM <= sl.p) { // LEFT FAN
    Real SHL = sl.u - sl.c ;
    if ( S <= SHL ) { //LEFT DATA STATE
      sm.r = sl.r ;
      sm.u = sl.u ;
      sm.p = sl.p ;
      sm.c = sl.c ;
    } else {
      Real CML = sl.c * pow(PM / sl.p, G1 ) ;
      Real STL = UM - CML ;
      if ( S > STL ) { // MIDDLE LEFT STATE
        sm.r = sl.r * pow(PM / sl.p, G8 ) ;
        sm.u = UM ;
        sm.p = PM ;
        sm.c = sqrt( GAMMA * sm.p / sm.r ) ;
      } else { // FAN LEFT STATE (INSIDE FAN)
        sm.u = G5 * (sl.c + G7 * sl.u + S)  ;
        sm.c = G5 * (sl.c + G7 * (sl.u - S) )   ;
        sm.r = sl.r * pow(sm.c / sl.c, G4 ) ;
        sm.p = sl.p * pow(sm.c / sl.c, G3 ) ;
      }
    }
  } else { // LEFT SHOCK
     Real PML = PM / sl.p ;
     Real SL = sl.u - sl.c * sqrt(G2 * PML + G1) ;
     if ( S <= SL ) { // LEFT DATA STATE
       sm.r = sl.r ;
       sm.u = sl.u ;
       sm.p = sl.p ;
       sm.c = sl.c ;
     } else { // MIDDLE LEFT STATE (BEHIND SHOCK)
       sm.r = sl.r * (PML + G6 ) / (PML * G6 + 1.0) ;
       sm.u = UM ;
       sm.p = PM ;
       sm.c = sqrt( GAMMA * sm.p / sm.r ) ;
     }
   }
 } else { // RIGHT OF CONTACT
   if ( PM > sr.p ) { // RIGHT SHOCK
```

```
        Real PMR = PM / sr.p ;
        Real SR  = sr.u + sr.c * sqrt(G2 * PMR + G1 ) ;
        if (S >= SR) { // RIGHT DATA STATE
          sm.r = sr.r ;
          sm.u = sr.u ;
          sm.p = sr.p ;
          sm.c = sr.c ;
        } else { // MIDDLE RIGHT STATE (BEHIND SHOCK)
          sm.r = sr.r * (PMR + G6 ) / (PMR * G6 + 1.0) ;
          sm.u = UM ;
          sm.p = PM ;
          sm.c = sqrt( GAMMA * sm.p / sm.r ) ;
        }
      } else { // RIGHT FAN
        Real SHR = sr.u + sr.c ;
        if ( S >= SHR ) { // RIGHT DATA STATE
          sm.r = sr.r ;
          sm.u = sr.u ;
          sm.p = sr.p ;
          sm.c = sr.c ;
        } else {
          Real CMR = sr.c * pow(PM / sr.p, G1 ) ;
          Real STR = UM + CMR ;
          if (S <= STR) { // MIDDLE RIGHT STATE
            sm.r = sr.r * pow(PM / sr.p, G8 ) ;
            sm.u = UM ;
            sm.p = PM ;
            sm.c = sqrt( GAMMA * sm.p / sm.r ) ;
          } else { // FAN RIGHT STATE (INSIDE FAN)
            sm.u = G5 * ( - sr.c + G7 * sr.u + S) ;
            sm.c = G5 * (sr.c - G7 * (sr.u - S) ) ;
            sm.r = sr.r * pow(sm.c / sr.c, G4 ) ;
            sm.p = sr.p * pow(sm.c / sr.c, G3 ) ;
          }
        }
      }
    }
}
```

References

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

[2] G.H.GOLUB, AND VAN LOAN, C. *Matrix Computations (second edition)*. Johns Hopkins University Press, 1990.

[3] GLASS, G., AND SCHUCHERT, B. *The STL <Primer>*. Prentice Hall PTR, 1995.

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

[5] MEISTER, A., AND SONAR, T. Finite-volume schemes for compressible fluid flow. *Surv. Math. Ind. 8* (1998), 1–36.

[6] MUSSER, D., AND SAINI, A. *STL tutorial & reference guide: C++ programming with the standard template library*. Addison-Wesley, 1996.

[7] STROUSTRUP, B. *The C++ Programming Language (Third Edition)*. Addison-Wesley, 1998.

[8] TORO, E. F. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer-Verlag, 1997.