Tools for the Solution of PDEs Defined on Curved Manifolds with the deal.II Library

Antonio DeSimone, Luca Heltai*, Cataldo Manigrasso

Abstract

The `deal.II` finite element library was originally designed to solve partial differential equations defined on one, two or three space dimensions, mostly via the Finite Element Method. In its versions prior to version 6.2, the user could not solve problems defined on curved manifolds embedded in two or three spacial dimensions. This infrastructure is needed if one wants to solve, for example, Boundary Integral Equations.

We present a collection of `C++` classes and utilities written by the authors, that were built on top of the existing `deal.II` library in order to extend its capabilities to the treatment of problems defined on manifolds of codimension one, such those arising from Boundary Element discretizations of Boundary Integral Equations.

We provide the reader with a detailed explanation of the building blocks that were added to the library, together with a complete example of collocation Boundary Element Method (included in release 6.2 of the `deal.II` library as `step-34`) that will allow the user to take full advantage of the new capabilities of `deal.II` .

*Corresponding Author. Email: Luca Heltai <luca.heltai@sissa.it>, Tel.: +39 040 3787449; Fax: +39 040 3787528

1 Introduction

The `deal.II` library is a collection of `C++` classes conceived to solve PDE problems formulated in the framework of the finite element method. It can handle problems in one, two or three dimensions using the most common finite elements spaces of arbitrary order, and it is fully customizable to account for user provided finite element spaces.

One of the main characteristics of the `deal.II` library is that the geometrical and topological information that describe the triangulation are kept separate from the logical data that describe the finite dimensional space, allowing for great flexibility in user codes.

All the relevant information is easily accessible through STL-like iterators that can cycle on cells, faces or edges. Many quadrature formulas (Gauss, Gauss-Lobatto, closed Newton-Cotes, iterated formulas, user-defined formulas, etc.) and mappings from the reference element to the one in real space (bi- and trilinear, polynomial of higher order, C^1 continous, Eulerian, Cartesian) are supported.

The library supports hp-adaptivity, multilevel methods and anisotropic refinements and it provides its own iterative solvers and preconditioners, as well as interfaces to efficient external libraries for the solution of linear systems via direct methods (UMFPACK, HSL) and to both serial and parallel libraries (PETSc, Trilinos) dedicated to the iterative solution of linear systems (see [2] for a brief introduction on the general framework of the `deal.II` library).

The library is thoroughly documented (see [3]) and it is equipped with a rich set of example programs (the documentation would be around 4000 pages if printed).

Version 6.2 of the `deal.II` library, among other enhancements and bug fixes, contains a set of tools which extends the capabilities of the `deal.II` library to the solution of PDEs defined on curved manifolds of codimension one with respect to the embedding space. The main motivation for our contribution was the resolution of Boundary Integral Equations on complex geometries, via the Boundary Element Method.

We explain in details the extensions that were made to the library in section 2. A complete example program [5] was added to the library to show the usage of these tools for the solution of irrotational flow around complex geometries. The example is analyzed in details in section 3.

We would like to thank W. Bangerth and the other maintainers of the

deal.II library for the precious help and for the numerous suggestions they offered us during our work.

We gratefully acknowledge reliable scientific interaction with "*Centro di Ricerca Navale*" (CETENA) and financial support from the *"consorzio RINAVE"*.

2 Codimension one modules

The basic ingredient for all partial differential equations is the definition of a domain, typically defined as a subset Ω of $\Re^n$, where n is normally one, two or three. A finite element discretization of Ω, as implemented in the `deal.II` library, requires one to approximate Ω with a collection of simple geometrical entities (called *cells* from here on) grouped together in an accessible data structure: the `Triangulation`.

Up to now the library could only handle n-dimensional triangulations in n-dimensional spaces. The declaration of the class used to describe such a collection of cells would look like:

```
...
Triangulation<dim> tria;
...
```

where the `dim` template parameter specifies the dimension of both the mesh and the space.

If one is interested in solving partial differential equations defined on curved manifolds, for example a curve in two dimensions or a surface in three dimensions (this is the case, for example, in Boundary Integral Equations, where the partial differential equation is expressed through quantities defined only on the *boundary* of the domain Ω) then the requirements for the data structure holding the cell information is different: one needs to describe (`dim`-1)-dimensional manifold in a `dim`-dimensional space.

It seemed natural to extend the above class for our needs, by constructing a generalized `Triangulation` class that would handle manifolds as in

```
...
Triangulation<dim, spacedim> tria;
...
```

where the `spacedim` parameter specifies the dimension of the geometrical space in which the mesh is embedded. Almost all classes of the library have been modified in this way, with the `spacedim` parameter taking the default value `dim` so that existing programs need not be changed.

2.1 Geometrical entities

The only data a `Triangulation` stores are the geometric and topological properties of a mesh: where vertices are located, and how these vertices are connected to cells. The properties and data of a `Triangulation` are almost always queried through loops over all cells. Most of the knowledge about a mesh is therefore hidden behind iterators, i.e., pointer-like structures that one can use to iterate from one cell to the next, and that one can query for information about the cell it presently points to.

Several programs are available, both open source and commercial ones, that can provide a discretization of codimension one curves or surfaces. The `deal.II` library offers an interface to read and write several mesh formats, but not all the formats which are supported for finite element meshes are compatible with codimension one problems.

The only reason for this limitation is given by the fact that vertex specification needs to be compatible with the euclidean structure of the embedding space. Currently the only two supported formats that support specifications of the vertex points in three dimensions (independent on the manifold dimension) are the UCD and GMSH formats.

We therefore modified the `GridIn` and `GridOut` classes of the library to allow reading and writing of the mesh information in the `spacedim`-dimensional environment.

A typical usage of these classes looks like the code below:

```
...
Triangulation<dim, spacedim> tria;
GridIn<dim, spacedim> gi;
gi.attach_triangulation (tria);
std::ifstream in (filename);
gi.read_ucd (in); // gi.read_msh (in);
...
```

where a `Triangulation` and a `GridIn` classes are instantiated. The first one is the container of the data structure of the mesh (empty upon con-

struction), while the second one is the helper class that enables us to fill the `Triangulation` data structure with the data of the new mesh (list of vertices and segments, quadrilaterals or hexahedra) read from a UCD (or, equivalently, GMSH) file.

Notice that, as for almost every program that uses `deal.II` , one can write a dimension-independent code thanks to the template parameters associated to the dimension and to the space dimension.

The complementary instructions that write the `Triangulation` data structure to a file make use of the class `GridOut`:

```
...
GridOut grid_out;
grid_out.write_ucd (tria, outfile_ucd);
grid_out.write_msh (tria, outfile_msh);
...
```

When we work with codimension one objects, there are not only normals to the faces of the elements as it is usual in the `deal.II` library, but also normals to the cells themselves. It is therefore important to number correctly the vertices of the cells, so to obtain normals pointing in the desired direction.

The implemented `GridIn` class respects the orientation of the original UCD and GMSH files. In these file formats, vertices are ordered either clockwise, or anticlockwise. We adopted the standard "right handed" orientation, that is, if we face an element whose vertices are oriented anticlockwise, the normal comes towards us (Figure 1).

While in standard finite element meshes this is not relevant, for boundary element methods it is of crucial importance that all elements are oriented in a consistent manner.

An example of mesh in the `UCD` format (extension `.inp`) is given by:

```
8 6 0 0 0
1  -0.577350269 -0.577350269 -0.577350269
2   0.577350269 -0.577350269 -0.577350269
3  -0.577350269  0.577350269 -0.577350269
4   0.577350269  0.577350269 -0.577350269
5  -0.577350269 -0.577350269  0.577350269
6   0.577350269 -0.577350269  0.577350269
7  -0.577350269  0.577350269  0.577350269
```

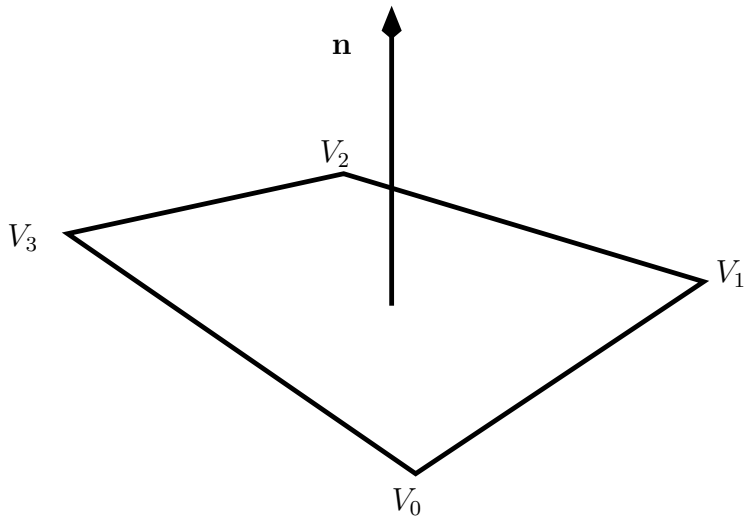

Figure 1: Orientation of the normals with respect to the vertices.

```
8   0.577350269  0.577350269  0.577350269
1 1 quad 3 4 2 1
2 1 quad 5 6 8 7
3 1 quad 1 2 6 5
4 1 quad 3 7 8 4
5 1 quad 5 7 3 1
6 1 quad 2 4 8 6
```

which describes the cube inscribed in the sphere of radius 1, with normals directed in the outward direction.

Once we create a mesh, it might be of use to perform some local or global refinement, in order to improve the accuracy of the computed solution. While in standard finite element codes this can be done by adding a vertex in the barycenter of the cells which are selected for refinement, for codimension one cells, the manifold that we want to approximate is seldom flat, and it is desirable that new vertices are added that lay on the original manifold, rather than on the approximated one.

To this end, the `deal.II` library provides a very similar feature for the boundary of finite element meshes in the class `Boundary<dim>`.

Whenever a cell at the boundary is refined, it is necessary to introduce at least one new vertex on the boundary. In the simplest case, one assumes that

the boundary consists of straight line segments between the vertices of the original, coarsest mesh, and the next vertex is simply put into the middle of the old ones. This is the default behavior of the `Triangulation<dim>` class, and is described by the `StraightBoundary<dim>` class.

On the other hand, if one deals with curved boundaries, this is not the appropriate thing to do. The classes derived from the `Boundary<dim>` base class therefore describe the geometry of a domain. One can then attach an object of a class derived from this base class to the `Triangulation<dim>` object using the `Triangulation<dim>::set_boundary()` function. Several classes exist to support the most common geometries.

We modified the behavior of the original `Boundary<dim>` class, to adapt it also to the cases where the triangulation describes a codimension one manifold.

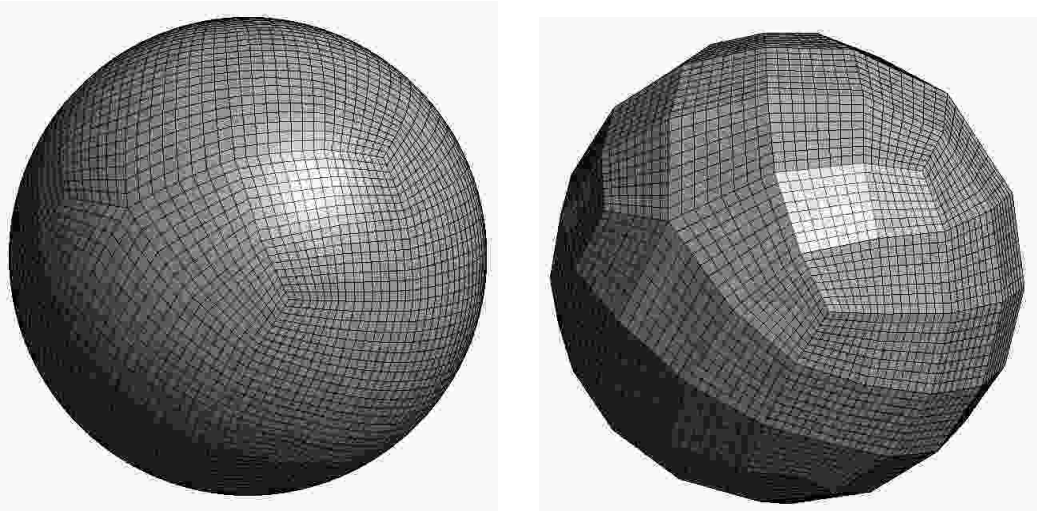

Figure 2: Comparison between mesh refinements with and without manifold description.

The use is exactly the same as in the standard case:

```
...
Point<spacedim> p(0,0,0);
// A spherical surface centered in p with radius 1
HyperBallBoundary<dim, spacedim> boundary(p,1);
```

```
Triangulation<dim, spacedim> tria;
tria.set_boundary(1, boundary);
...
```

The different results in case we use or we do not use the boundary description can be seen in Figure 2.

2.2 Basis functions and transformations

Once we have imported and approximated the topological domain, the next step is to define the degrees of freedom associated with the vector space defined on the approximated domain itself. In the `deal.II` library this is obtained through the synergic use of the classes `DoFHandler<dim>` and `FiniteElement<dim>`.

Finite element classes describe the properties of a finite element space as defined on the unit cell. This includes, for example, how many degrees of freedom are located at vertices, on lines, or in the interior of cells. In addition to this, finite element classes of course have to provide values and gradients of individual shape functions at points on the unit cell.

`DoFHandler` objects are the confluence of triangulations and finite elements: the finite element class describes how many degrees of freedom it needs per vertex, line, or cell, and the `DoFHandler` class allocates this space so that each vertex, line, or cell of the triangulation has the correct number of them and it gives them a global numbering.

Just as with triangulation objects, most operations on `DoFHandler`s is done by looping over all cells and doing something on each or a subset of them. The interfaces of the two classes are therefore rather similar: they allow to get iterators to the first and last cell (or face, or line, etc.) and offer information through these iterators. The information that can be extracted from these iterators is the geometric and topological information that can already be extracted from the triangulation iterators (they are in fact derived classes) as well as things like the global numbers of the degrees of freedom on the present cell.

Among the various possibilities offered by the `deal.II` library, we limited the current implementation of the codimension-one modules to the following specializations of the `FiniteElement<dim,spacedim>` class:

- `FE_Q<dim,spacedim>`: continuous tensor product polynomial space of arbitrary order;

- `FE_DGQ<dim,spacedim>`: discontinuous tensor product polynomial space of arbitrary order;
- `FE_DGP<dim,spacedim>`: discontinuous polynomial space of arbitrary order.

The usage of these classes follows very closely the standard `deal.II` procedures (see, e.g. [2]):

```
Triangulation<dim,spacedim> tria;
FE_Q<dim,spacedim> fe(1);
DoFHandler<dim,spacedim> dh(tria);
dh.distribute_dofs(fe);
...
```

At the end of the above snippet of code, the object `dh` will contain the data structures needed to approximate and work with a continuous piecewise bilinear vector space, defined on the manifold approximated by the triangulation `tria`.

In particular we can use the tools provided by the `deal.II` library to define a vector of scalar values that uniquely identifies an object of the given vector space:

```
...
DoFHandler<dim,spacedim> dh(tria);
...
Vector<double> solution(dh.n_dofs());
...
```

The vector `solution` has dimension `dh.n_dofs()`, equal to the number of degrees of freedom that identifies the vector space `dh`.

The next step in a finite or boundary element program is that one would want to compute matrix and right hand side entries or other quantities on each cell of a triangulation, using the shape functions of a finite element and quadrature points defined by a quadrature rule.

To this end, it is necessary to map the shape functions, quadrature points, and quadrature weights from the unit cell to each cell of a triangulation. This is facilitated by the `Mapping` and derived classes: they describe how to map points from the unit cell to the real space and back, as well as provide gradients of this transformation and Jacobian determinants.

For the moment we implemented the `MappingQ1<dim,spacedim>` and `MappingQ1Eulerian<dim,spacedim>` class, which extend the functionality of the original `deal.II` classes to the codimension one case.

```
...
QTrapez<dim> quad;
MappingQ1<dim,spacedim> mapping;
typename Triangulation<dim,spacedim>::active_cell_iterator
  cell=tria.begin_active(),
  endc=tria.end() ;
Point<spacedim> real;
Point<dim> unit;
for(;cell!=endc;++cell)
  {
    deallog<<cell<< std::endl;
    for(unsigned int q=0; q<quad.n_quadrature_points; ++q)
      {
        real = mapping.transform_unit_to_real_cell
              (cell, quad.point(q));
        std::cout <<quad.point(q)<< " -> " << real << std::endl;
      }
  }
```

The above snippet of code simply prints the vertices of each cell of the triangulation, which corresponds to the mapped quadrature points of a trapezoidal quadrature integration rule in two dimensions from the unit cell (the points $(0,0)$, $(1,0)$, $(0,1)$ and $(1,1)$) to the real cell.

The next step is to actually take a finite element and evaluate its shape functions and their gradients at the points defined by a quadrature formula when mapped to the real cell, which is done by the `FEValues` class.

For example, to compute the total area of a circular or spherical mesh, together with the error in the computation of the outer normals to the cells, one could use the following snippet of code:

```
...
const QGauss<dim> quadrature(2);
const FE_Q<dim,spacedim> fe (1);
DoFHandler<dim,spacedim> dof_handler (triangulation);
```

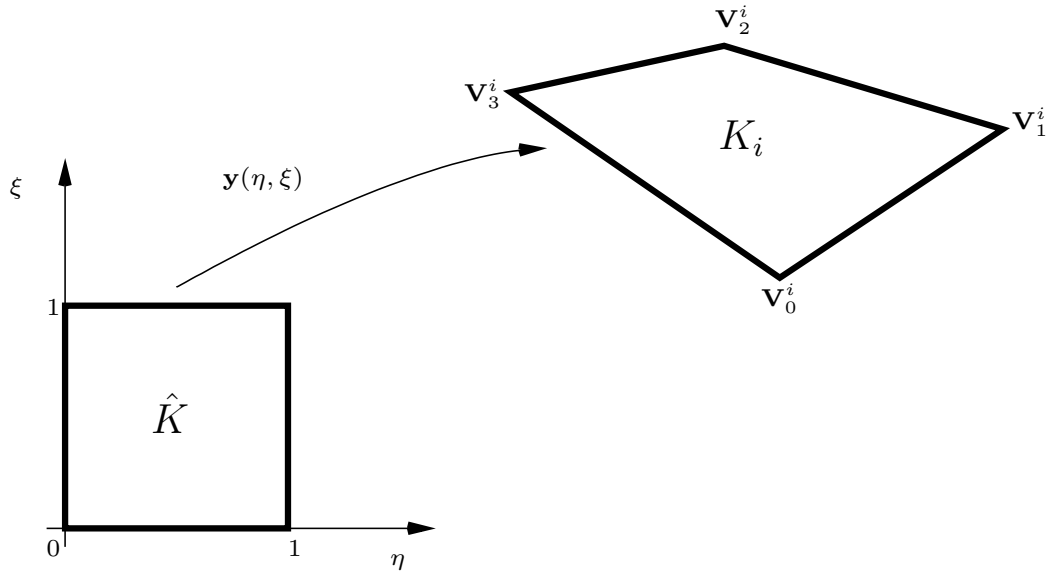

Figure 3: Transformation from reference to real cell.

```
FEValues<dim,spacedim> fe_values
    (dummy_fe, quadrature,
     update_JxW_values |
     update_cell_normal_vectors |
     update_quadrature_points);

dof_handler.distribute_dofs (fe);

double area = 0;
double normals = 0;

typename DoFHandler<dim,spacedim>::active_cell_iterator
    cell = dof_handler.begin_active(),
    endc = dof_handler.end();

std::vector<Point<spacedim> >
    expectedcellnormals(fe_values.n_quadrature_points);

for (; cell!=endc; ++cell)
```

```
{
    fe_values.reinit (cell);
    const std::vector<Point<spacedim> >
            & cellnormals =
               fe_values.get_cell_normal_vectors();
    const std::vector<Point<spacedim> >
            & quad_points =
               fe_values.get_quadrature_points();

    for (unsigned int i=0;
         i<fe_values.n_quadrature_points;
         ++i)
    {
        expectedcellnormals[i] =
            quad_points[i]/quad_points[i].norm();
        area += fe_values.JxW (i);
        normals +=
          (expectedcellnormals[i]-cellnormals[i]).norm();
    }
};
```

Notice that in order to compute the outer normals, we introduced a new flag for the `FEValues` classes, named `update_cell_normal_vectors`, which instructs the `FEValues` class to compute and store the normals to the cells.

2.3 Interpolation, projection and graphical output

Once we constructed the above tools, the entire machinery is in place that allows us the use of existent `deal.II` functions and utilities. An example is the interpolation and projection of continuous scalar and vector fields on the finite dimensional spaces defined above.

The `deal.II` library provided already some utilities to interpolate and project arbitrary functions on Finite Element spaces. We adapted these functions to be used also with the codimension one versions, such that the following snippet of code can be used, pretty much in the same way that was previously available to `deal.II` users:

```
...
```

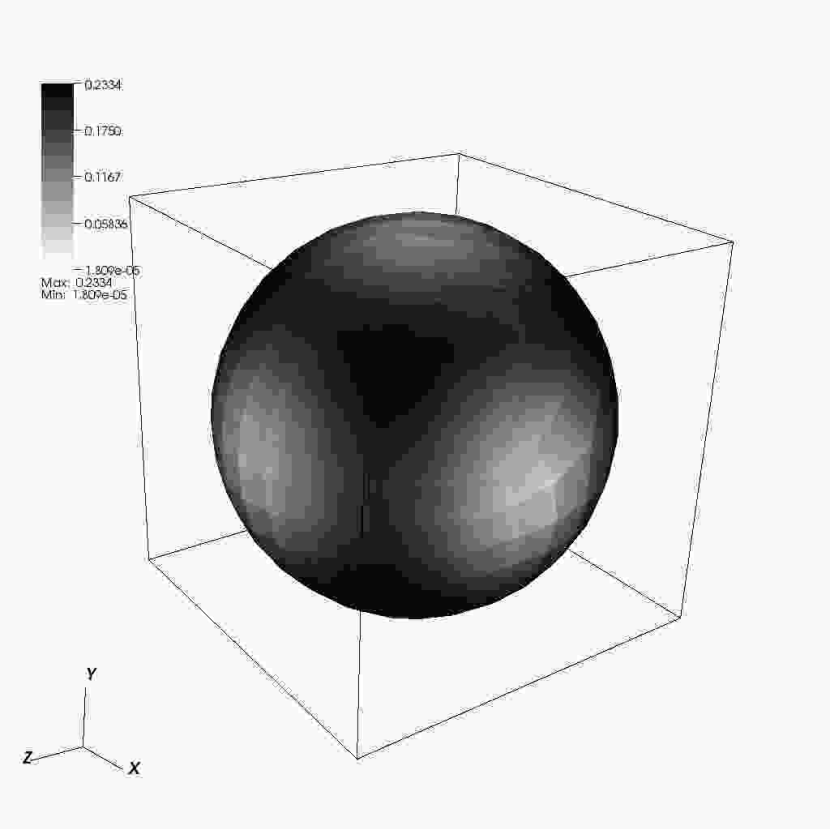

Figure 4: Projection of the cosine function on the surface of a sphere.

```
Vector<double> projected_one(dof_handler.n_dofs());

Functions::CosineFunction<spacedim> cosine;
QGauss<dim> quad(5);
ConstraintMatrix constraints;
constraints.close();

VectorTools::project
      (dof_handler,
       constraints,
       quad,
       cosine,
       projected_one);
```

```
VectorTools::interpolate
      (dof_handler,
       constant,
       interpolated_one);
```

The result of the given example can be seen in Figure 4, which was realized using the class `DataOut`. Again, this class was modified to accept the new codimension one objects without changing the user interface:

```
DataOut<dim, DoFHandler<dim,spacedim> > dataout;
dataout.attach_dof_handler(dof_handler);
dataout.add_data_vector(projected_one, "projection");
dataout.build_patches();
std::ofstream file("output.vtk");
dataout.write_vtk(file);
```

3 A complete example

We wrote a complete example to show in great details the usage of all the added tools in a concrete case ([5]) which has been included in version 6.2 of the `deal.II` library. Here we describe some details of its implementation which allows one to solve the problem of irrotational fluid flowing around a circular or a spherical obstacle in two or three spacial dimensions respectively.

3.1 Mathematical overview

The motion of an inviscid fluid past a body at rest is usually modeled by the Euler equations of fluid dynamics:

$$\begin{cases} \frac{\partial}{\partial t}\mathbf{v} + (\mathbf{v}\cdot\nabla)\mathbf{v} &= -\frac{1}{\rho}\nabla p + \mathbf{g} & \text{in } \mathbb{R}^n\backslash\Omega \\ \nabla\cdot\mathbf{v} &= 0 & \text{in } \mathbb{R}^n\backslash\Omega \end{cases}, \tag{1}$$

where the fluid density ρ and the acceleration $\mathbf{g}$ due to external forces are given, while the velocity $\mathbf{v}$ and the pressure p are the unknowns. Here Ω is a closed bounded region representing the body around which the fluid moves and we assume that the boundary $\Gamma = \partial\Omega$ is sufficiently smooth.

Uniqueness of the solution is ensured by the following boundary conditions

$$\begin{aligned} \mathbf{n} \cdot \mathbf{v} &= 0 \qquad && \text{on } \partial\Omega \\ \mathbf{v} &= \mathbf{v}_\infty \qquad && \text{when } |\mathbf{x}| \to \infty, \end{aligned} \tag{2}$$

which mean that the body is not permeable, and that the fluid is assumed in uniform motion at infinity.

Equations (1) can be derived from Navier-Stokes equations assuming that the effects due to viscosity are negligible compared to those due to the pressure gradient and to the external forces.

We will concentrate only on the stationary solutions to problem (1). These solutions are typically used to understand the behavior of the given (possibly complex) geometry when a fluid moves around it or, equivalently, when a prescribed motion, e.g., a translation, is enforced on the body. This model can describe for example the motion of air past an airplane wing, or, with some trivial modifications, the motion of air or water past a propeller rotating at constant speed.

For both stationary and non stationary flow, the solution is sought by solving firstly for the velocity in the second equation and then by substituting in the first equation in order to find the pressure. It is convenient to decompose the velocity of the fluid in a uniform background field $\mathbf{v}_\infty$ and in a perturbation field $\mathbf{v_p}$ which is due to the presence of the body Ω

$$\nabla \cdot \mathbf{v} = \nabla \cdot (\mathbf{v}_\infty + \mathbf{v_p}) = \nabla \cdot \mathbf{v_p} = 0.$$

If we assume that the fluid is irrotational, i.e., $\nabla \times \mathbf{v} = 0$ in $\mathbb{R}^n \backslash \Omega$, we can represent the velocity, and consequently also the perturbation velocity, as the gradient of a scalar function:

$$\mathbf{v_p} = \nabla\phi,$$

which implies that the second of equations (1) above can be rewritten as Laplace equation for the unknown ϕ with Neumann boundary conditions on the boundary $\partial\Omega$

$$\Delta\phi = 0 \qquad \text{in } \Omega$$

$$\mathbf{n} \cdot \nabla\phi = -\mathbf{n} \cdot \mathbf{v}_\infty \qquad \text{on } \partial\Omega \tag{3}$$

$$\phi = \phi_\infty \qquad \text{when } |\mathbf{x}| \to \infty, \tag{4}$$

where the last condition ensures unicity. The conservation of momentum equation (the first of equations (1)) reduces to Bernoulli's equation that expresses the pressure p as a function of the potential ϕ:

$$\frac{p}{\rho} + \frac{1}{2}|\nabla\phi|^2 = 0 \in \Omega.$$

Notice that, by assuming that the flow is irrotational, we completely decoupled the conservation of momentum from the conservation of mass, and the two problems can be solved one after the other. For what follows, we will only be interested in solving for the unknown potential ϕ, since the pressure can be easily recovered by postprocessing the solution ϕ.

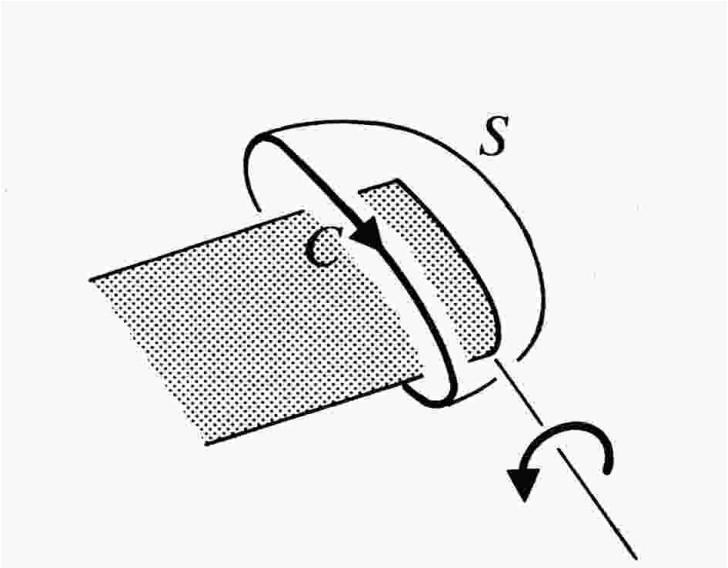

Figure 5: Three dimensional irrotational flow cannot generate lift.

The irrotationality assumption is not without consequences, since it affects in a significant way the properties of the solutions. The Kutta-Jukowski theorem (see, e.g., chapter 4 of [1]), states that a flux can generate lift only if the circulation $\oint_C \mathbf{v} \cdot d\mathbf{s}$ on any closed curve C around the body is different from zero. Using Stoke's theorem this implies that

$$\oint_C \mathbf{v} \cdot d\mathbf{s} = \int_S \nabla \times \mathbf{v}\, d\mathbf{x}$$

where S is any surface with C as boundary.

In two dimensions the integral above vanishes if S does not contain Ω, but can be different from zero otherwise, which implies that Euler equations can describe a lifting flux.

On the other hand, in the three dimensional case, which is the most interesting for practical applications, the irrotationality of $\mathbf{v}$ in $\mathbb{R}^3\backslash\Omega$ makes any circulation around a finite body vanish, since for any circuit there is always an associated surface completely contained in $\mathbb{R}^3\backslash\Omega$ (Fig.5 from [1]). One way to fix this problem is to add artificially some sources of vorticity around the body on the so called trailing wake, so that lift can be generated. For more information on this see chapter 6 of [7]. Notice that we show an example without a wake, although no additional tools need to be added to the library in order to treat the more general case.

Following [4], we reformulate in integral form the Laplace equation above using the second Green identity:

$$\int_{\mathbb{R}^n\backslash\Omega}(-\Delta u)v\,dx-\int_{\partial\Omega\cup\Gamma_\infty}\frac{\partial u}{\partial\mathbf{n}}v\,ds=\int_{\mathbb{R}^n\backslash\Omega}(-\Delta v)u\,dx-\int_{\partial\Omega\cup\Gamma_\infty}u\frac{\partial v}{\partial\mathbf{n}}\,ds \quad (5)$$

where $\mathbf{n}$ is the normal to $\partial\Omega$ pointing towards the fluid $\mathbb{R}^n\backslash\Omega$. Notice that the integral operator $\int_{\Gamma_\infty} ds$ should be interpreted in the following sense:

$$\int_{\Gamma_\infty} ds := \lim_{r\to\infty}\int_{\partial B_r(0)} ds.$$

We remind the reader that the following functions, called fundamental solutions of Laplace equation,

$$G(\mathbf{x}-\mathbf{y}) = -\frac{1}{2\pi}\ln|\mathbf{x}-\mathbf{y}| \qquad \text{for } n=2 \quad (6)$$

$$G(\mathbf{x}-\mathbf{y}) = \frac{1}{4\pi}\frac{1}{|\mathbf{x}-\mathbf{y}|} \qquad \text{for } n=3, \quad (7)$$

satisfy in variational sense

$$-\Delta_x G(\mathbf{x}-\mathbf{y}) = \delta(\mathbf{x}-\mathbf{y})$$

where the differentiation is done with respect to $\mathbf{x}$. If we substitute u and v in (5) with the solution ϕ and with the fundamental solution G respectively,

we obtain for all $\mathbf{x} \in \mathbb{R}^n \backslash \Omega$

$$\begin{aligned}\phi(\mathbf{x}) &= -\int_{\partial\Omega\cup\Gamma\infty} G(\mathbf{x}-\mathbf{y})\frac{\partial\phi}{\partial\mathbf{n}_y}(\mathbf{y})\,ds_y + \int_{\partial\Omega\cup\Gamma\infty}\frac{\partial G(\mathbf{x}-\mathbf{y})}{\partial\mathbf{n}_y}\phi(\mathbf{y})\,ds_y \\ &= \phi_\infty - \int_{\partial\Omega} G(\mathbf{x}-\mathbf{y})\frac{\partial\phi}{\partial\mathbf{n}_y}(\mathbf{y})\,ds_y + \int_{\partial\Omega}\frac{\partial G(\mathbf{x}-\mathbf{y})}{\partial\mathbf{n}_y}\phi(\mathbf{y})\,ds_y. \qquad (8)\end{aligned}$$

In the above was used the fact that, since $\lim_{\mathbf{x}\to\infty}\phi(\mathbf{x}) = \phi_\infty \in \mathbb{R}$, one has to consider only the term

$$\int_{\Gamma\infty}\frac{\partial G(\mathbf{x}-\mathbf{y})}{\partial\mathbf{n}_y}\phi(\mathbf{y})\,ds_y = -\lim_{r\to\infty}\int_{\partial B_r(0)}\frac{\mathbf{r}}{r}\nabla G(\mathbf{x}-\mathbf{y})\phi_\infty\,ds_y = \phi_\infty.$$

Equation (8) can be rewritten in a more compact form using the Single Layer Potential (SLP) and the Double Layer Potential (DLP) operators:

$$\phi(\mathbf{x}) = \phi_\infty - \left(S\frac{\partial\phi}{\partial\mathbf{n_y}}\right)(\mathbf{x}) + (D\phi)(\mathbf{x}) \quad \text{for all } \mathbf{x}\in\mathbb{R}^n\backslash\Omega. \qquad (9)$$

It can be shown that (3) and (9) are equivalent [4]. Notice also that the last equation allows one to calculate ϕ in any point of the domain once its expression on the boundary $\partial\Omega$ is known.

If we impose the boundary conditions (2) we obtain:

$$\mathbf{n}\cdot(\mathbf{v}_\infty + \mathbf{v}_p) = 0 \quad \Rightarrow \quad \mathbf{n}\cdot\mathbf{v_p} = -\mathbf{n}\cdot\mathbf{v}_\infty \quad \Rightarrow \quad \frac{\partial\phi}{\partial\mathbf{n}} = -\mathbf{n}\cdot\mathbf{v}_\infty.$$

Moreover (see, for example, [8]) if we take the limit for $\mathbf{x}$ tending to $\partial\Omega$ in equation (9) we can reduce it to an expression on the boundary only:

$$\alpha(\mathbf{x})\phi(\mathbf{x}) = \phi_\infty + \left(S[\mathbf{n}\cdot\mathbf{v}_\infty]\right)(\mathbf{x}) + (D\phi)(\mathbf{x}) \quad \text{for all } \mathbf{x}\in\partial\Omega, \qquad (10)$$

which is the integral formulation we were looking for. The quantity $\alpha(\mathbf{x})$ is the fraction of angle or of solid angle by which the point $\mathbf{x}$ sees the fluid $\mathbb{R}^n\backslash\Omega$. In particular, at points $\mathbf{x}$ where the boundary $\partial\Omega$ is differentiable we have $\alpha(\mathbf{x}) = 1/2$, but the value may be different at points where the boundary has a corner or an edge.

Substituting the explicit expressions of SLP and DLP we can rewrite equation (10) as:

$$\alpha(\mathbf{x})\phi(\mathbf{x}) = \phi_\infty - \frac{1}{2\pi}\int_{\partial\Omega} \ln|\mathbf{x}-\mathbf{y}|\,\mathbf{n}\cdot\mathbf{v}_\infty\,ds_y + \frac{1}{2\pi}\int_{\partial\Omega}\frac{(\mathbf{x}-\mathbf{y})\cdot\mathbf{n}_y}{|\mathbf{x}-\mathbf{y}|^2}\phi(\mathbf{y})\;ds_y \tag{11}$$

$$\alpha(\mathbf{x})\phi(\mathbf{x}) = \phi_\infty + \frac{1}{4\pi}\int_{\partial\Omega}\frac{1}{|\mathbf{x}-\mathbf{y}|}\,\mathbf{n}\cdot\mathbf{v}_\infty\,ds_y + \frac{1}{4\pi}\int_{\partial\Omega}\frac{(\mathbf{x}-\mathbf{y})\cdot\mathbf{n}_y}{|\mathbf{x}-\mathbf{y}|^3}\phi(\mathbf{y})\,ds_y \tag{12}$$

for two dimensional and for three dimensional flows respectively.

Since the constant function $\phi(\mathbf{x}) = \phi_\infty$ is a solution of the Laplace equation with zero Neumann data, the following expression

$$\alpha(\mathbf{x}) := 1 - \frac{1}{2(n-1)\pi}\int_{\partial\Omega}\frac{(\mathbf{y}-\mathbf{x})\cdot\mathbf{n}_y}{|\mathbf{y}-\mathbf{x}|^n}\phi(\mathbf{y})\,ds_y = 1 + \int_{\partial\Omega}\frac{\partial G(\mathbf{y}-\mathbf{x})}{\partial\mathbf{n}_y}\,ds_y,$$

can be used to derive explicitly the values of $\alpha(\mathbf{x})$ on $\partial\Omega$.

3.2 Numerical approximation

Numerical approximations of boundary integral equations are commonly referred to as the Boundary Element Method or Panel Method (the latter expression being used mostly in the computational fluid dynamics community [6]). This naming convention derives from the use of quadrilateral meshes for the discretization of the computational domain in panels.

The goal of the following test problem is to solve the integral formulation of the Laplace equation described in the previous section, using a circle and a sphere respectively in two and three space dimensions. To this end, let $\mathcal{T}_h = \bigcup_i^M K_i$ be a partition of the manifold $\Gamma = \partial\Omega$ into M line segments if $n = 2$, or M quadrilaterals if $n = 3$. We will call each individual K_i element or cell, independently of the dimension of the space and we require that the partition, more frequently referred to as triangulation, be regular in the sense explained in [10]. We define the finite dimensional space V_h as

$$V_h := \{v \in C^0(\Gamma) \text{ s.t. } v|_{K_i} \in \mathcal{Q}^1(K), \forall K \in \mathcal{T}_h\}$$

where $\mathcal{Q}^1(K)$ indicates the space of polynomials of maximum degree 1 in each variable on the domain K. An element ϕ_h of V_h is uniquely determined

by the vector $\boldsymbol{\phi}$ of its coefficients ϕ_i, that is:

$$\phi_h(\mathbf{x}) := \sum_{i=1}^{N} \phi_i \psi_i(\mathbf{x}), \qquad \boldsymbol{\phi} := \{\phi_i\},$$

where $\{\psi_i\}$ is the set of basis functions that has cardinality $N = \dim V_h$. The reader is reminded that each of these functions is associated to a vertex of the triangulation, so that it takes the value 1 on one vertex and vanishes on all the others:

$$\psi_i(\mathbf{x}_j) = \delta_{ij}. \tag{13}$$

By far, the most common approximation of boundary integral equations in the engineering community is by use of the collocation boundary element method. This method requires that the boundary integral equation be satisfied at a number of collocation points which is equal to the number of unknowns of the system, namely the N coefficients that identify the solution ϕ_h. The choice of these points is a delicate matter, but let us assume for the moment that their are known, and call them $\mathbf{x}_i$ with $i = 0...N$.

If we consider equation (10), we obtain the following problem. Given the datum $\mathbf{v}_\infty$, find a function ϕ_h in V_h such that the N equations

$$\alpha(\mathbf{x}_i)\phi_h(\mathbf{x}_i) - \int_{\Gamma_y} \frac{\partial G(\mathbf{y} - \mathbf{x}_i)}{\partial \mathbf{n}_y} \phi_h(\mathbf{y})\, ds_y = \int_{\Gamma_y} G(\mathbf{y} - \mathbf{x}_i)\, \mathbf{n}_y \cdot \mathbf{v}_\infty\, ds_y,$$

are satisfied, where we have set the arbitrary constant ϕ_∞ in (10) to zero. The problem can then be written as the following linear system:

$$(\mathbf{A} + \mathbf{N})\boldsymbol{\phi} = \mathbf{b}$$

where

$$\begin{aligned}
\mathbf{A}_{ij} &= \alpha(\mathbf{x}_i)\psi_j(\mathbf{x}_i) = 1 + \int_{\Gamma} \frac{\partial G(\mathbf{y} - \mathbf{x}_i)}{\partial \mathbf{n}_y}\, ds_y \psi_j(\mathbf{x}_i) \\
\mathbf{N}_{ij} &= -\int_{\Gamma} \frac{\partial G(\mathbf{y} - \mathbf{x}_i)}{\partial \mathbf{n}_y} \psi_j(\mathbf{y})\, ds_y \\
\mathbf{b}_i &= \int_{\Gamma} G(\mathbf{y} - \mathbf{x}_i)\, \mathbf{n}_y \cdot \mathbf{v}_\infty ds_y.
\end{aligned}$$

From a linear algebra point of view, the best possible choice of the collocation points is the one that renders the matrix $\mathbf{A}$+$\mathbf{N}$ the most diagonally dominant. A natural choice is then to select the $\mathbf{x}_i$ collocation points to be the vertices

associated to the nodal basis functions $\psi_i(\mathbf{x})$. As a consequence of (13), the matrix $\mathbf{A}$ is diagonal with entries

$$\mathbf{A}_{ii} = 1 + \int_\Gamma \frac{\partial G(\mathbf{y} - \mathbf{x}_i)}{\partial \mathbf{n}_y}\, ds_y = 1 - \sum_j N_{ij}$$

where we have used that $\sum_j \psi_j(\mathbf{y}) = 1$ for the usual Lagrange elements. With this choice of collocation points, the computation of the entries of the matrices $\mathbf{A}$, $\mathbf{N}$ and of the right hand side $\mathbf{b}$ requires the evaluation of singular integrals on the elements K_i of the triangulation $\mathcal{T}_h$. As usual in these cases, all integrations are performed on a planar reference domain, i.e., we assume that each element K_i of $\mathcal{T}_h$ can be expressed as a linear (in two dimensions) or bi-linear (in three dimensions) transformation of the reference boundary element $\hat{K} := [0,1]^{n-1}$ (cf. Figure 3). We then perform the integrations after a change of variables from the real element K_i to the reference element $\hat{K}$.

In two dimensions the diagonal elements $\mathbf{N}_{ii}$ of the system matrix vanish because $\mathbf{n}_y$ and $(\mathbf{y} - \mathbf{x})$ are orthogonal if $\partial\Omega$ is approximated with segments (cf. equations (11) and (12)). The only singular integral arises in the computation of $\mathbf{b}_i$ on the i-th element of $\mathcal{T}_h$:

$$\frac{1}{\pi} \int_{K_i} \ln |\mathbf{y} - \mathbf{x}_i|\, \mathbf{n}_y \cdot \mathbf{v}_\infty\, ds_y,$$

which can be anyway treated with special Gauss integration formulas like the ones proposed in [9].

We treat similarly the singularities of type $1/r$ appearing in the three dimensional case, following [8]. The main idea of these integration formulas is to eliminate the singularity using an appropriate coordinate transformation. We split the integration domain $\hat{K} = (0,1)^2$ in two triangles and consider the lower one $T_L = \mathrm{conv}\{(0,0),(1,0),(1,1)\}$. The transformation

$$\begin{aligned} &F : (0,1)^2 \to T_L, \\ &(u,v) \mapsto \left(u, u \tan\left(\frac{\pi}{4} v\right)\right), \end{aligned}$$

whose jacobian is

$$J = \frac{\pi}{4} \frac{r}{\cos(\frac{\pi}{4} v)},$$

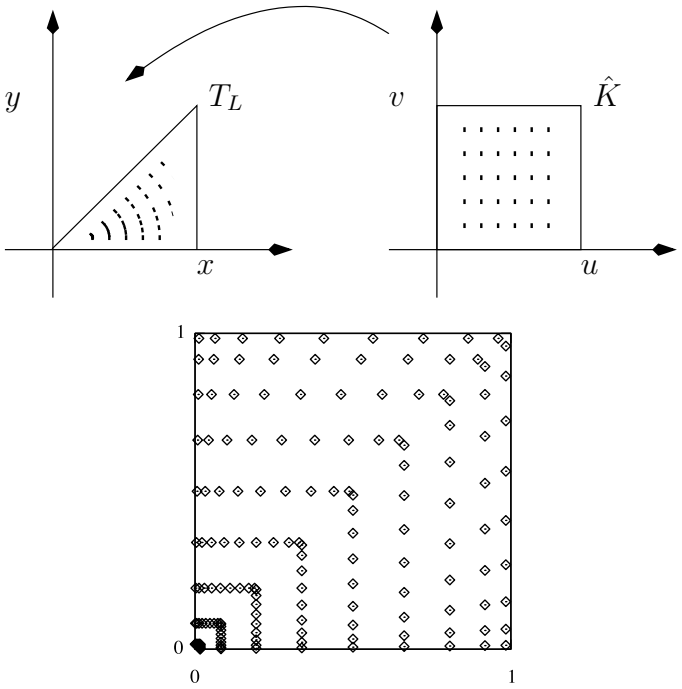

Figure 6: Coordinate transformation for the treatment of singularities in the three dimensional case.

introduces a term proportional to r into the integral. This term cancels out the singularity in the integrand and we can use the ordinary Gauss integration formulas (Figure 6). The upper triangle in the reference element $\hat{K}$ can be then treated analogously.

The resulting matrix $\mathbf{A} + \mathbf{N}$ is full. Depending on its size, it might be convenient to use a direct solver or an iterative one. For the purpose of this example code, we chose to use only an iterative solver, without providing any preconditioner.

If this were an industrial project rather than a demonstration of the new functionalities added to `deal.II` , the need for precision would soon require a large number of panels and the solution of a linear system of the type described above with a large number of unknowns would not be feasible. In this case more sophisticated techniques are available that allow one not to store full matrices by neglecting higher order terms due to panels far away from each other. In the literature on boundary element methods a

plethory of these methods have been developed, leading to a significantly sparser representation of these matrices that also facilitates rapid evaluation of vector-matrix products ([10], [8]).

3.3 Implementation

We will now discuss the actual implementation of the problem, an example of which is the tutorial `step-34` in the `deal.II` library [5]. In order to understand this example, the reader is recommended to practice with the `deal.II` framework using the large set of tutorials that are distributed with the library. We suggest to read at least the first five tutorials to get familiar with the basic steps in the implementation of a problem.

The main structure of the program is implemented in the `BEMProblem` class. This structure should be already familiar to `deal.II` users, as well as to any Finite Element code programmers, as it is almost self explanatory:

```
template <int dim>
class BEMProblem
{
  public:
    BEMProblem();

    void run();

  private:

    void read_parameters (const std::string &filename);

    void read_domain();

    void refine_and_resize();

    void assemble_system();

    void solve_system();

    void compute_errors(const unsigned int cycle);
```

```
    void compute_exterior_solution();

    void output_results(const unsigned int cycle);
};
```

The only really new function that we find here is the assembly routine. We wrote this function in the most possible general way, in order to allow for easy generalization to higher order methods and to different fundamental solutions (e.g., Stokes or Maxwell). The most noticeable difference from FEM codes is the fact that the final matrix is full, and that we have a nested loop inside the usual loop on cells that visits all the points associated to the degrees of freedom. Moreover, when the point lies inside the cell which is being visited, the integrand becomes singular and needs a special treatment. The practical consequence is that we have two sets of quadrature formulas, finite element values and temporary data, one for standard integration and one for the singular integration.

We opted for an iterative solver in the function `solve_system()`. The matrix that we assemble is full and not symmetric, and we chose the unpreconditioned GMRES method. The options for the iterative solver, such as the tolerance and the maximum number of iterations are selected through the parameter file (cf. the function `read_parameters()`).

Once we obtained the solution, we compute the L^2 error of the potential as well as the L^∞ error of the approximation of the solid angle in `compute_errors()`. The mesh we are using is an approximation of a smooth curve or surface, therefore the computed diagonal matrix of fraction of angles or solid angles $\alpha(\mathbf{x})$ should contain entries that approximate the exact value $1/2$.

Once a solution on the codimension one domain is obtained, we would like to extend it to the rest of the external space $\mathbb{R}^n \backslash \Omega$. This is done by performing the convolution of the solution with the kernel in the `compute_exterior_solution()` function, following equation (8). The velocity field can then be easily obtained computing the gradient of the potential. All the results are plotted by the function `output_results()`.

The objects that will be shared by the functions of the class are then declared specifying the optional argument that indicates the dimension of the space the body is embedded in.

```
    Triangulation<dim-1,dim> tria;
```

```
    FE_Q<dim-1,dim>          fe;
    DoFHandler<dim-1,dim>    dh;

    FullMatrix<double>       system_matrix;
    Vector<double>           system_rhs;

    Vector<double>           phi;
    Vector<double>           alpha;

    ConvergenceTable         convergence_table;

    Functions::ParsedFunction<dim> wind;
    Functions::ParsedFunction<dim> exact_solution;

    std_cxx1x::shared_ptr<Quadrature<dim-1> > quadrature;
    unsigned int singular_quadrature_order;

    SolverControl solver_control;

    unsigned int n_cycles;
    unsigned int external_refinement;

    bool run_in_this_dimension;
    bool extend_solution;
};
```

Notice that, since the assembling process will produce a full system matrix, we cannot use a sparse matrix here as in the usual FEM codes. The convergence table is used to output errors in the exact solution and in the computed alphas.

The `Functions::ParsedFunction` class allows us to easily and quickly define new function objects via parameter files, with custom definitions which can be very complex (see the documentation of that class int the `deal.II` guide for all the available options). We will allocate the quadrature object using the `QuadratureSelector` class that allows us to generate quadrature formulas based on an identifying string and on the possible degree of the formula itself. We also define a couple of parameters which are used in case we want to extend the solution to the entire domain.

In the following we will give an outline of the most important parts of the program (we refer the reader to [5] for the complete code and for further explanations).

3.3.1 Assembling the system matrix

The main difficulty encountered in this routine is the treatment of the singular integrals. At the beginning of this function, we create the singular quadrature formulas for the three dimensional problem. Here we define a vector of four such quadratures (one per each vertex of a cell) that will be used later on.

```
template <int dim>
void BEMProblem<dim>::assemble_system()
{
  std::vector<QGaussOneOverR<2> > sing_quadratures_3d;
  for(unsigned int i=0; i<4; ++i)
    sing_quadratures_3d.push_back
      (QGaussOneOverR<2>(singular_quadrature_order, i, true));
```

Next, we initialize a `FEValues` object with the quadrature formula for the integration of the kernel in non singular cells. This quadrature is selected with the parameter file, and needs to be quite precise, since the functions we are integrating are not polynomial functions.

```
  FEValues<dim-1,dim> fe_v(fe, *quadrature,
                           update_values |
                           update_cell_normal_vectors |
                           update_quadrature_points |
                           update_JxW_values);

  const unsigned int n_q_points = fe_v.n_quadrature_points;

  std::vector<unsigned int> local_dof_indices(fe.dofs_per_cell);

  std::vector<Vector<double> > cell_wind(n_q_points, Vector<double>(dim) );
  double normal_wind;
```

Unlike in finite element methods, if we use collocation BEM, then in each assembly loop we only assemble the information that refers to the coupling

between one degree of freedom (the degree associated with vertex `i`) and the current cell.

We can start the integration loop over all cells, where we first initialize the FEValues object and get the values of $\mathbf{v_p}$ at the quadrature points:

```
typename DoFHandler<dim-1,dim>::active_cell_iterator
  cell = dh.begin_active(),
  endc = dh.end();

for(cell = dh.begin_active(); cell != endc; ++cell)
  {
    fe_v.reinit(cell);
    cell->get_dof_indices(local_dof_indices);

    const std::vector<Point<dim> > &q_points =
             fe_v.get_quadrature_points();
    const std::vector<Point<dim> > &normals =
             fe_v.get_cell_normal_vectors();
    wind.vector_value_list(q_points, cell_wind);
```

We then compute the integral over the current cell for all degrees of freedom (note that this includes also degrees of freedom not located on the current cell, a deviation from the usual finite element integrals). The integral that we need to perform is singular if one of the local degrees of freedom is the same as the support point `i`. If the index `i` is not one of the local degrees of freedom, we simply have to add the single layer terms to the right hand side, and the double layer terms to the matrix:

```
if(is_singular == false)
  {
   for(unsigned int q=0; q<n_q_points; ++q)
    {
     normal_wind = 0;
     for(unsigned int d=0; d<dim; ++d)
     normal_wind += normals[q][d]*cell_wind[q](d);

     const Point<dim> R = q_points[q] - support_points[i];

     system_rhs(i) += ( LaplaceKernel::single_layer(R)*
```

```
                        normal_wind*
                        fe_v.JxW(q) );

    for(unsigned int j=0; j<fe.dofs_per_cell; ++j)
     local_matrix_row_i(j) -= ( ( LaplaceKernel::double_layer(R)*
                                  normals[q] )*
                                  fe_v.shape_value(j,q)*
                                  fe_v.JxW(q)       );
   }
 } else {
```

Now we treat the more delicate case. In this case both the single and the double layer potential are singular, and they require special treatment as described earlier. In one dimension this process does not result only in a factor appearing as a constant factor on the entire integral, but also on an additional integral alltogether that needs to be evaluated:

$$\int_0^1 f(x)\ln(x/\alpha)dx = \int_0^1 f(x)\ln(x)dx - \int_0^1 f(x)\ln(\alpha)dx.$$

This process is taken care of by the constructor of the `QGaussLogR` class, which adds additional quadrature points and weights to take into consideration also the second part of the integral.

Putting all this into a dimension independent framework requires a little trick. The problem is that, depending on the dimension, we would like to either assign a `QGaussLogR<1>` or a `QGaussOneOverR<2>` to a `Quadrature<dim-1>`. C++ does not allow this right away, and neither is a `static_cast` possible but we are allowed to do a `dynamic_cast`.

```
const Quadrature<dim-1> *
 singular_quadrature
   = (dim == 2?
      dynamic_cast<Quadrature<dim-1>*>(
      new QGaussLogR<1>(singular_quadrature_order,
                        Point<1>((double)singular_index),
                        1./cell->measure(), true))
      :
      (dim == 3?
       dynamic_cast<Quadrature<dim-1>*>(
```

```
         &sing_quadratures_3d[singular_index])
        :0));

  Assert(singular_quadrature, ExcInternalError());
...
  for(unsigned int q=0; q<singular_quadrature->size(); ++q)
   {
    const Point<dim> R = singular_q_points[q] - support_points[i];
    double normal_wind = 0;
    for(unsigned int d=0; d<dim; ++d)
     normal_wind += (singular_cell_wind[q](d)*
                     singular_normals[q][d]);

    system_rhs(i) += ( LaplaceKernel::single_layer(R) *
                       normal_wind*
                       fe_v_singular.JxW(q) );

    for(unsigned int j=0; j<fe.dofs_per_cell; ++j)
     {
      local_matrix_row_i(j) -= (( LaplaceKernel::double_layer(R) *
                                  singular_normals[q])*
                                  fe_v_singular.shape_value(j,q)*
                                  fe_v_singular.JxW(q) );
     }
   }
  if(dim==2)
   delete singular_quadrature;
 }
```

Finally, we need to add the contributions of the current cell to the global matrix.

```
  for(unsigned int j=0; j<fe.dofs_per_cell; ++j)
   system_matrix(i,local_dof_indices[j])
   += local_matrix_row_i(j);
```

The second part of the integral operator is the term $\alpha(\mathbf{x}_i)\phi_j(\mathbf{x}_i)$. Since we use a collocation scheme, $\phi_j(\mathbf{x}_i) = \delta_{ij}$ and the corresponding matrix is a diagonal one with entries equal to $\alpha(\mathbf{x}_i)$. One quick way to compute this diagonal matrix of the solid angles, is to use the DLP matrix itself.

```
  Vector<double> ones(dh.n_dofs());
  ones.add(-1.);

  system_matrix.vmult(alpha, ones);
  alpha.add(1);
  for(unsigned int i = 0; i<dh.n_dofs(); ++i)
      system_matrix(i,i) +=  alpha(i);
}
```

3.3.2 Extending the solution

With this function we compute finally the value of the potential ϕ in the exterior domain $\mathbb{R}^n \backslash \Omega$, considering that our original problem consist in knowing the velocity field in the fluid.

To this end, we extrapolate the actual solution inside the box $[-2, 2]^{\dim}$ using the convolution with the fundamental solution. The reconstruction of the solution in the entire space is done on a continuous finite element grid of dimension `dim`.

```
template <int dim>
void BEMProblem<dim>::compute_exterior_solution()
{
  Triangulation<dim>  external_tria;
  GridGenerator::hyper_cube(external_tria, -2, 2);

  FE_Q<dim>           external_fe(1);
  DoFHandler<dim>     external_dh (external_tria);
  Vector<double>      external_phi;
...
  typename DoFHandler<dim-1,dim>::active_cell_iterator
    cell = dh.begin_active(),
    endc = dh.end();

  FEValues<dim-1,dim> fe_v(fe, *quadrature,
                           update_values |
                           update_cell_normal_vectors |
                           update_quadrature_points |
                           update_JxW_values);
```

```
...
   typename DoFHandler<dim>::active_cell_iterator
...
   std::vector<Point<dim> > external_support_points
                (external_dh.n_dofs());
   DoFTools::map_dofs_to_support_points<dim>
        ( StaticMappingQ1<dim>::mapping,
          external_dh, external_support_points);

   for(cell = dh.begin_active(); cell != endc; ++cell)
     {
...
       for(unsigned int i=0; i<external_dh.n_dofs(); ++i)
         for(unsigned int q=0; q<n_q_points; ++q)
           {

             const Point<dim> R =
                q_points[q] - external_support_points[i];

             external_phi(i) +=
                ( ( LaplaceKernel::single_layer(R) *
                    normal_wind[q]
                    +
                    (LaplaceKernel::double_layer(R) *
                     normals[q] )            *
                    local_phi[q] )           *
                  fe_v.JxW(q) );
           }
     }
...
 }
```

3.3.3 Numerical results and solution plotsr

As we can see from the convergence table in two dimensions, if we choose quadrature formulas which are accurate enough, then the error we obtain for $\alpha(\mathbf{x})$ should be exactly the inverse of the number of elements. The approximation of the circle with N segments of equal size generates a regular

polygon with N faces, whose angles are exactly $\pi - \frac{2\pi}{N}$, therefore the error we commit should be exactly $1/2 - (1/2 - 1/N) = 1/N$. In fact this is a very good indicator that we are performing the singular integrals in an appropiate manner.

The error in the approximation of the potential ϕ is largely due to approximation of the domain. A much better approximation could be obtained by using higher order mappings.

cycle	cells	dofs	L^2 (ϕ)	rate	L^∞ (α)	rate
0	20	20	4.465e-02	-	5.000e-02	-
1	40	40	1.081e-02	2.05	2.500e-02	1.00
2	80	80	2.644e-03	2.03	1.250e-02	1.00
3	160	160	6.529e-04	2.02	6.250e-03	1.00

cycle	cells	dofs	L^2 (ϕ)	rate	L^∞ (α)	rate
0	24	26	6.873e-01	-	2.327e-01	-
1	96	98	1.960e-01	1.81	1.239e-01	0.91
2	384	386	4.837e-02	2.02	6.319e-02	0.97
3	1536	1538	1.176e-02	2.04	3.176e-02	0.99

The result of running these computations is a bunch of output files that we can pass to our visualization program of choice. The output files are of two kind: the potential on the boundary element surface, and the potential extended to the outer and inner domain. The combination of the two for the two dimensional case can be observed in Figure 7.

For the three dimensional case, on the other hand, a better visualisation is obtained considering the potential first on the boundary of the sphere and then in the exterior region (Figure 8).

References

[1] D J Acheson. *Elementary Fluid Dynamics*. Oxford University Press, 1990.

[2] W. Bangerth, R. Hartmann, and G. Kanschat. deal.ii–a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24, 2007.

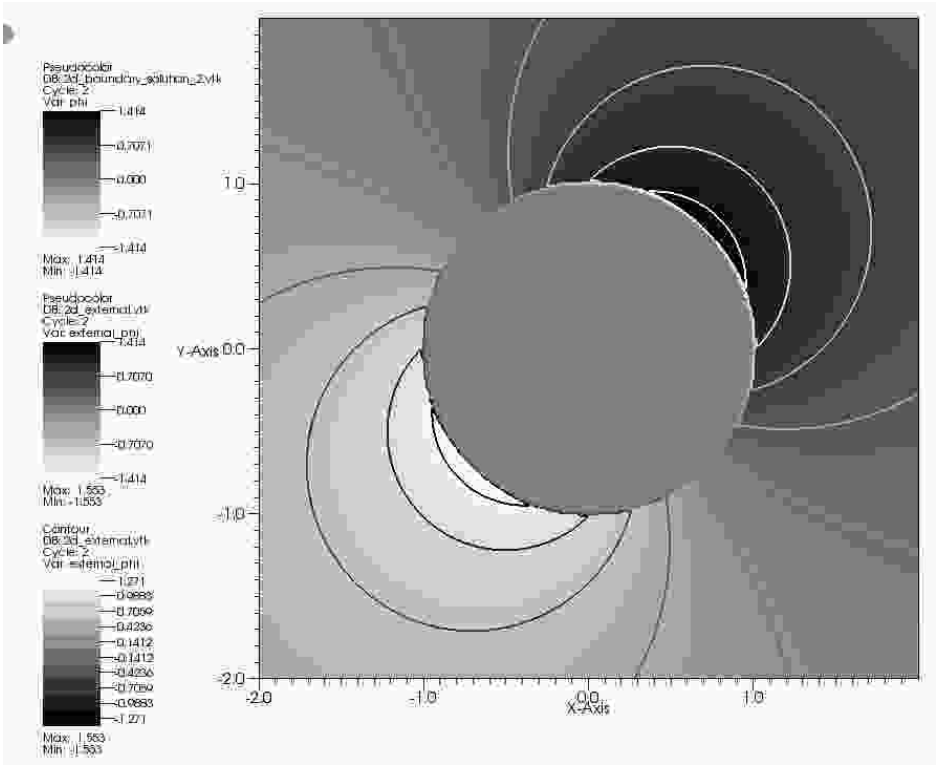

Figure 7: Potential ϕ on the boundary of the circle and in the exterior region.

[3] W Bangerth, Hartmann R, and Kanschat G. *deal.II Technical Reference* . http://www.dealii.org, 2009.

[4] L. Banjai. Boundary Element Methods. Course notes, 2007.

[5] L. Heltai. *Step 34,* `deal.II` *tutorial.* http://www.dealii.org/developer/doxygen/deal.II/step_34.html, 2009.

[6] J L Hess. Panel Methods in Computational Fluid Dynamics. *Annual Reviews in Fluid Mechanics*, 22(1):255–274, 1990.

[7] C. Marchioro and M. Pulvirenti. *Mathematical Theory of Incompressible Nonviscous Fluids.* Springer, 1994.

[8] Andrea Mola. *Models for Olympic Rowing Boats.* PhD thesis, MOX - Politecnico di Milano, 2009.

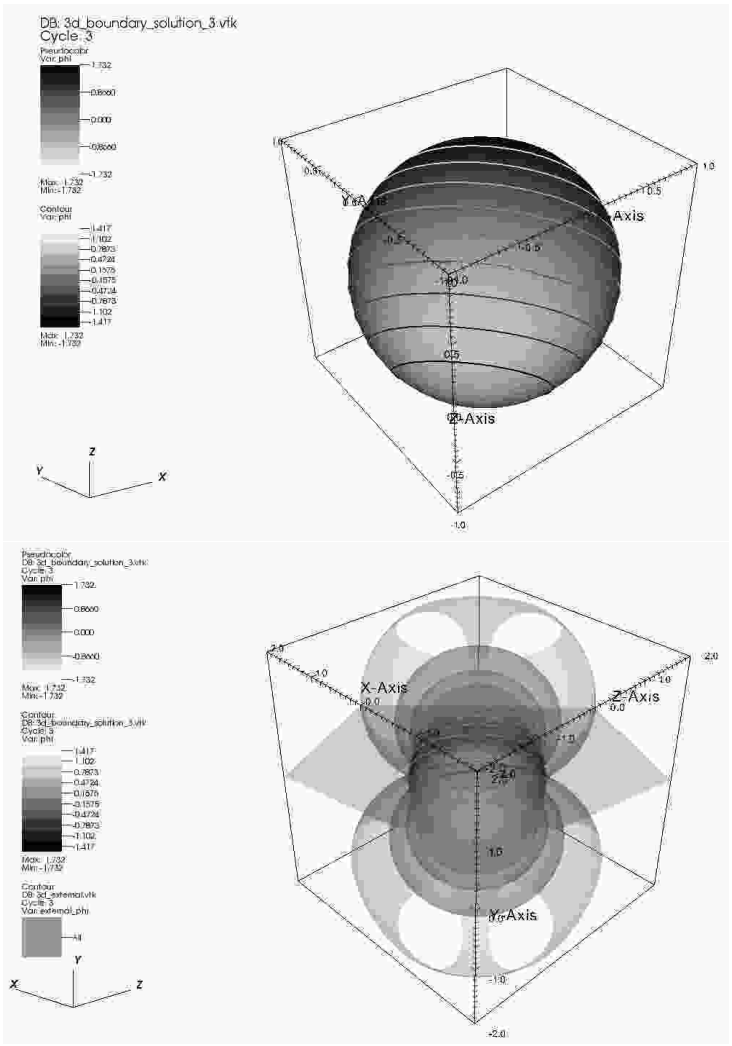

Figure 8: Potential ϕ on the boundary of the sphere and in the exterior region.

[9] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[10] S. Sauter and C. Schwab. *Randelementmethoden*. Teubner, 2004.