6
Computational Framework

In this chapter we describe important aspects of our finite element code. The aim of the chapter is to discuss these aspects, and not to comment the technical details. In Section 6.1 we propose a new data structure for mixed meshes with triangular and quadrangular elements called ECHE that is scalable in size, based on integer containers and integer arithmetic rules. In Section 6.2 we describe the implementation of the basic finite element framework used in Chapter 3. Finally, in Section 6.3, we discuss the extension of our code to handle fictitious domain simulations of flows with suspended particles.

PUC-Rio - Certificação Digital Nº 0610743/CA

6.1
Scalable topological data–structure for triangles and quadrangles meshes

The ECHE (*Extended Compact Half–Edge*) data structure extends the *Compact Half–Edge* (28) and at the same time can be considered a concise version of the *Handle–Edge* (29), since it can represent mixed meshes with triangles and/or quadrangles. It uses generic containers instead of pointers or static arrays. Similarly to the *Corner–Table* (35), it explicitly represents a few adjacency and incidence relations between the elements and uses a set of integer arithmetic rules to obtain the others.

The ECHE data structure actually has three levels, each one completing the previous in order to accelerate the execution time, but consuming a little more memory. It uses the concept of *half–edge* (see Figure 6.1) to represent the association of a face with one of its bounding edges, or equivalently the association of this edge with one of its vertices. Any access to the elements of a face is performed through its half–edges that are denoted by he.

Level 0: The level 0 of the ECHE represents only a soup of triangles and quadrangles by storing the starting vertex of each half–edge. The half–edges, the vertices and the faces are indexed by non–negative integers. Each face is represented, according to its type, by 3 or 4 consecutive half–edges that define its orientation. The indexing rules for the half–edges are the following: all half–edges belonging to faces of the same type comes in sequence, and triangle half–edges come before the quadrangle ones.

PUC-Rio - Certificação Digital Nº 0610743/CA

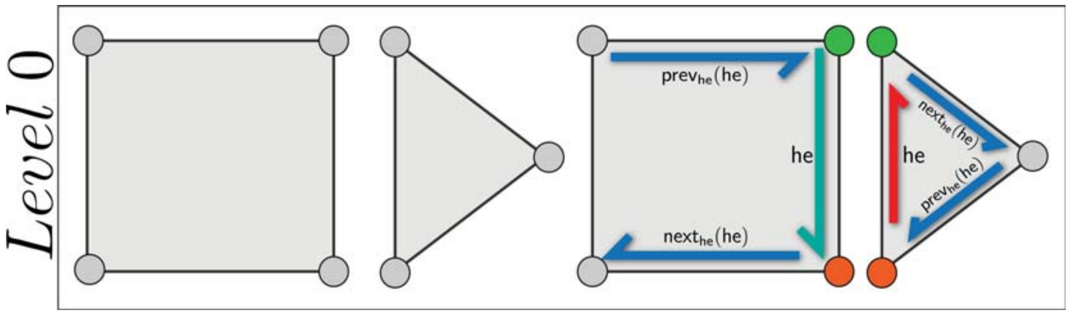

Figure 6.1: Level 0 of ECHE. Only a soup of triangles and quadrangles is represented (left). It is based on the concept of half edge (right). A half–edge is the association of a face with one of its bounding edges. In a given element the next and the previous half–edges are obtained using integer arithmetic rules.

The association of each half-edge he to its starting vertex is stored in one of the two containers of integers, named Tri–Vertex and Quad–Vertex containers and denoted by $\mathsf{V_t}[]$ and $\mathsf{V_q}[]$. If he is a triangle half–edge, the integer $\mathsf{v} = \mathsf{V_t}[\mathsf{he}]$ is the index of its starting vertex, when he is a quadrangle half–edge, its starting vertex is given by $\mathsf{v} = \mathsf{V_q}[\mathsf{he} - 3n_t]$. The size of $\mathsf{V_t}[]$ is $3n_t$ while $\mathsf{V_q}[]$ has $4n_q$ entries, where n_t and n_q denote, respectively, the number of triangles and quadrangles on the mesh.

Given a half–edge with index he, the first half–edge inside the face of he has index:

$$\mathsf{fbase_{he}}(\mathsf{he}) = \begin{cases} 3\lfloor \mathsf{he}/3 \rfloor & \text{when } \mathsf{he} < 3n_t \\ 4\lfloor (\mathsf{he} - 3n_t)/4 \rfloor + 3n_t & \text{otherwise} \end{cases}$$

Therefore, the indexes of the three half–edges that belong to the triangle with index t are 3t, 3t + 1, and 3t + 2. The indexes of the four half–edges of a quadrangle with index q are $4\mathsf{q} + 3n_t$, $4\mathsf{q} + 3n_t + 1$, $4\mathsf{q} + 3n_t + 2$ and $4\mathsf{q} + 3n_t + 3$. The next and previous half–edges of a given half–edge he on its associated element can be obtained by the use of the following rules and are shown in blue in Figure 6.1:

$$\begin{aligned} \mathsf{next_{he}}(\mathsf{he}) \;&:= \mathsf{fbase_{he}}(\mathsf{he}) + (\bar{\mathsf{he}} + \mathsf{ftype_{he}}(\mathsf{he}) \qquad)\%\mathsf{ftype_{he}}(\mathsf{he}), \\ \mathsf{prev_{he}}(\mathsf{he}) \;&:= \mathsf{fbase_{he}}(\mathsf{he}) + (\bar{\mathsf{he}} + \mathsf{ftype_{he}}(\mathsf{he}) - 1)\%\mathsf{ftype_{he}}(\mathsf{he}). \end{aligned}$$

where $\mathsf{ftype_{he}}(\mathsf{he}) = 3$ if $\mathsf{he} < 3n_t$ and 4 otherwise. We also use the notation, $\bar{\mathsf{he}}$ to represent the index of the half–edge given the geometry of its incident element, that is: $\bar{\mathsf{he}} = \mathsf{he}$ if $\mathsf{he} < 3n_t$ and $\bar{\mathsf{he}} = \mathsf{he} - 3n_t$ otherwise.

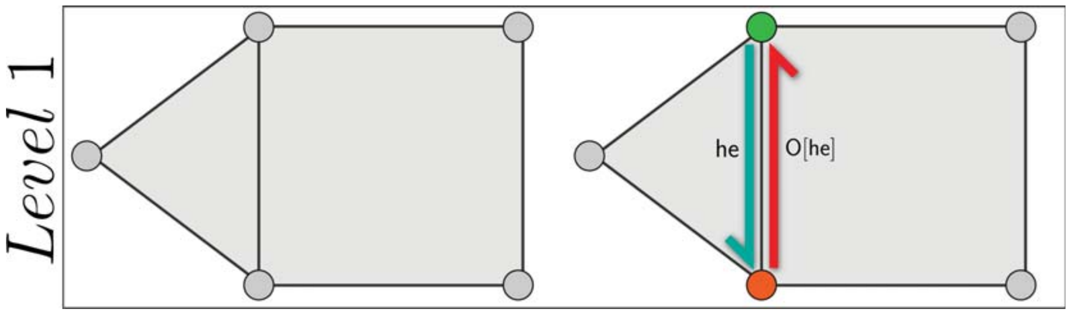

Figure 6.2: Level 1 of ECHE. Adds to level 0 the adjacency information of each element (left). The adjacency is encoded using the container of opposite half–edges. Two half–edges are opposites when they have the same vertices, but with opposite orientation (right).

Level 1: The level 1 of the ECHE adds to the level 0 the adjacency information of each element. Since we are working with 2–manifold, each half–edge is incident to one or two elements (see figure 6.2). In order to explicitly represent the adjacency relation of two elements, the ECHE uses another container of integers, named the Opposite container, denoted by O[].

The edge–adjacency between neighboring elements is represented by associating to each half–edge he its opposite half–edge O[he], which has the same vertices but opposite orientation. If the half–edge he is on the boundary, then it doesn't have an opposite, which is encoded by O[he] $= -1$. Thus, the value of O[he] allows to directly check whether a half–edge he is on the boundary or not. The size of O[] is $3n_t + 4n_q$.

Level 2: The level 2 of ECHE extends the Level 1 adding explicit representation for the vertices, edges and faces of the mesh (see Figure 6.3).

It is useful to store a new container that we call Extra vertex container and denote by VH[]. To compute simple geometry operators such as derivation, it is necessary to obtain the star of a vertex efficiently (we will define the vertex star later). Therefore, we can store on VH[] an integer that for each vertex v associates an index of the lower half–edge incident to vertex v. In the case the vertex is on the boundary, the stored half–edge should be the boundary one. Such container has size n_v, where n_v is the number of vertices on the mesh.

Moreover, incidence relations on edges are essential in many applications such as simplification and subdivision algorithms in computer graphics or finite elements in scientific computing. An edge is identified by its half–edge of lower index. The edges can be explicitly represented by a map called Edge map denoted by EH[], which maps an edge to the index of the lower of its two incident half–edges, and eventually to its attributes such as color, collapse cost and finite element's related degrees of freedom. The map EH[] has n_e entries, where n_e is the number of edges on the mesh.

PUC-Rio - Certificação Digital Nº 0610743/CA

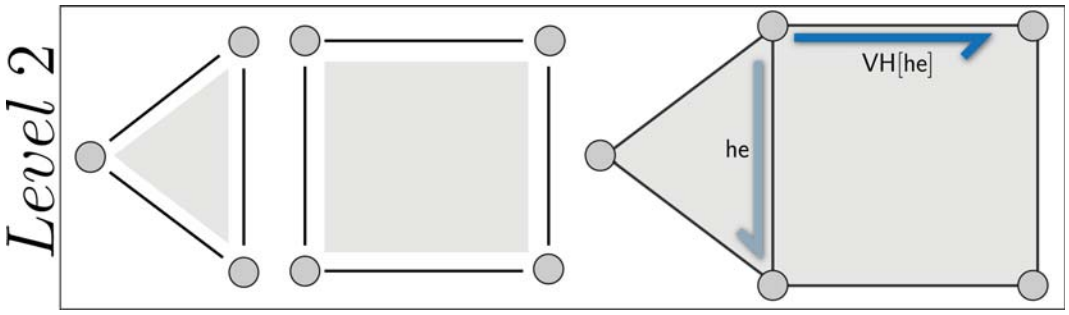

Figure 6.3: Third level of ECHE. Adds to the level 1 explicit representation of edges and faces of the mesh (left). To perform topological queries efficiently, the index of one incident half–edge is stored for each vertex of the mesh, as the blue boundary half–edge in the example (right).

Finally, in finite element, it is also useful to store an additional container called Face container and denoted by FH[] to represent the unknowns defined inside the faces. The map FH[] has n_f entries, where n_f is the number of elements on the mesh.

Using the proposed data structure, we can develop, in each level, topological procedures to transverse the mesh elements. The efficiency of those algorithms depends on the chosen ECHE level. When using the first levels, the computational performance decreases in order to get memory storage capability. On the other hand, working on higher levels costs more in terms of memory but reduces the execution time of topological queries. Those levels are implicit to the users by virtual inheritance: an object-oriented programming feature that avoids the programmer to care about which structure level is being used. With this resource, the topological queries always have the same interface, however their implementations may change at each new level.

Let us denote the topological query that returns the elements incident to a vertex by R_{02}, which is also known as the *vertex star* and is essential in the search structure that we propose in Section 6.3.

The ECHE answers relations R_{02} in time $O(n_t + n_q)$ at level 0, since the function has to transverse all the $\mathsf{V_t}[]$ and $\mathsf{V_q}[]$ containers. At level 1, the $\mathsf{V_t}[]$ and $\mathsf{V_q}[]$ containers are traversed until one half–edge incident to the input vertex v is found, after that the vertex star is obtained in time $O(deg(\mathsf{v}))$ by the use of the O[] and the rules described above. Thus, the worst case at level 1 has complexity $O(n_t+n_q)$, but it is in average $deg(\mathsf{v})$ times faster than for level 0. Finally, at level 2 the complexity of finding the star of a vertex v is reduced to $O(deg(\mathsf{v}))$, since VH[] directly stores the starting half–edge to traverse the vertex star. The algorithm to compute the R_{02} topological operation in the last ECHE level is described in Algorithm 2. More details on the implementation and efficiency of those topological queries are found in (28).

PUC-Rio - Certificação Digital Nº 0610743/CA

PUC-Rio - Certificação Digital Nº 0610743/CA

Algorithm 2 The vertex star

```
he $\leftarrow$ VH[v] //Gets the first incident half–edge
he$_0$ $\leftarrow$ he //Auxiliary half–edge
repeat
  he $\leftarrow$ next$_{\mathsf{he}}$(O[he])
  $R_{02}.push$(face$_{he}$(he))
until he$_0 \neq$ he OR O[he] $= -1$
```

6.1.1 ECHE example

In this sub–section we show explicitly the containers of the ECHE using a very simple example. The chosen mesh is the planning of a pyramid with squared base and without one of its lateral faces. In this simple example we can observe all features of ECHE in all its levels (see Figure 6.4).

In level 0, the containers $\mathsf{V_t}$[] and $\mathsf{V_q}$[] store the initial vertex of each half–edge belonging to triangles and quadrangles respectively. In level 1, the container O[] is added to the data–structure, and stores the opposite half–edges. Observe that the opposite of the boundary half–edges (he $= 1$, he $= 8$, he $= 12$) are set to -1. In the last level, we represent the first half–edge that parts from each vertex on the VH[] container. If we observe the entry VH[0] we see that, as the vertex v $= 0$ is a boundary vertex, its associated half–edge is the one with id he $= 1$ since it is the boundary half–edge parting from the vertex v $= 0$. Finally, we show the container EH[]. For each edge, we get its lower half–edge to be its identifier. When the edge is at the boundary, its identifier is its single half–edge, as we see in the edge with index he $= 1$.

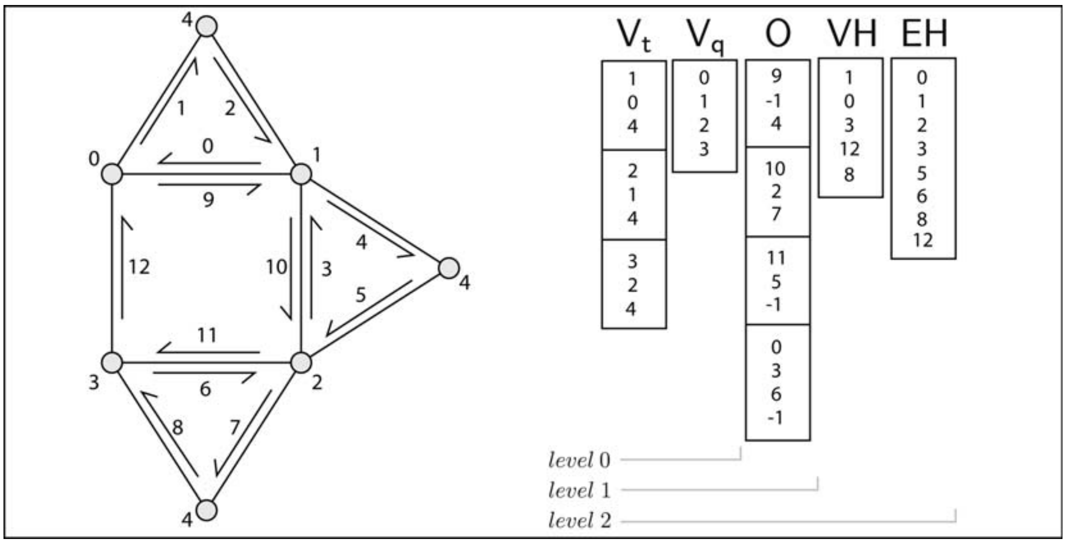

Figure 6.4: Simple example of ECHE. The left image shows a mesh with its half–edges while the right image shows the ECHE containers in each level.

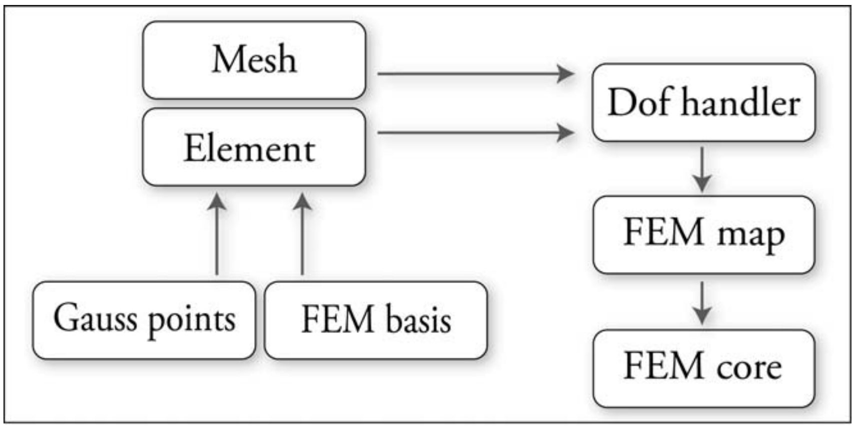

Figure 6.5: An overview of the relation between the most important classes in our code. The code description in this section follows this scheme.

6.2 Finite element code

Our finite element code was developed using the C++ programming language and inspired by the work of (4) where the authors describe structuring concepts and programming paradigms used to develop the finite element library called *deal.II*. By means of the structuring capabilities of C++, the different objects used in such a finite element simulation program are well separated. In particular, the separation of meshes, finite element spaces, and linear algebra classes allows for a very modular approach in programming finite element codes.

Although we use concepts proposed on the *deal.ii* library, the goals of our code are quite different. We did not intend to develop a general propose finite element library. Our aim was to have a fictitious domain finite element application using Lagrange multipliers based on the formulation proposed in Chapters 4 and 5 and use this code to study flows with suspended particles, that can float at the interface between immiscible fluids.

An overview of the relationship between the most important classes of our code for Newtonian incompressible flows is shown in Figure 6.5. The implementation description in this section follows this figure from left to right.

Gauss points, FEM basis and Element classes: In our code, the class Element manages the degrees of freedom in a given element. More precisely, the local indexation of the vertices, edges and face nodes of given a triangle or quadrangle are stored in this class. The adopted codification for the nodes is shown in Figure 6.6.

The indexation of degrees of freedom is closely related with the choice of the finite element approximation space used to compute the unknowns of the problem. For this reason, the Element class inherits from the class called FEM

PUC-Rio - Certificação Digital Nº 0610743/CA

PUC-Rio - Certificação Digital Nº 0610743/CA

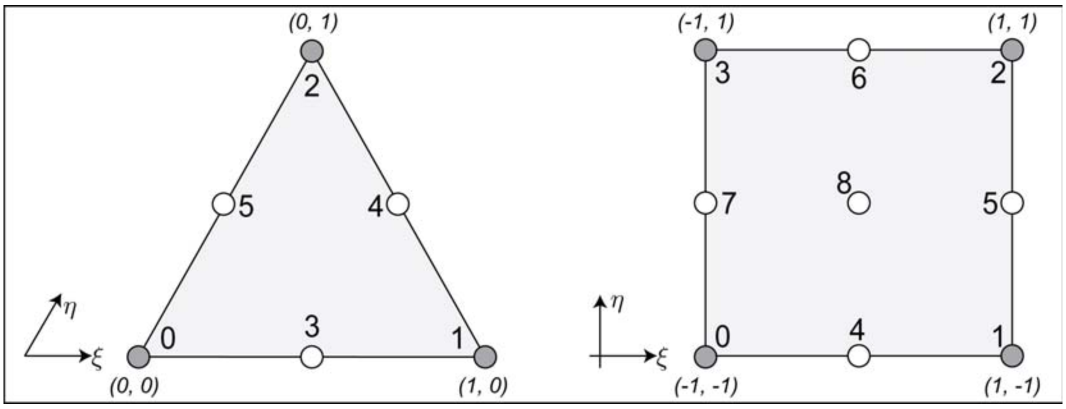

Figure 6.6: The local indexation and the local system of coordinates (ξ, η) adopted in the code are shown. The gray and white nodes are used to build quadratic and biquadratic basis, while for bilinear and linear finite elements spaces only the gray ones are used.

basis, the basis function's definition and also its derivatives computed on the local coordinates (ξ, η) defined inside each element. Figure 6.6 shows the local coordinate system for each type of element.

In our code, we use biquadratic and bilinear basis function for quadrangular elements and quadratic and linear functions for triangular elements. Figure 6.6 shows, in gray, the nodes used to build the bilinear (for quadrangles) and linear (for triangles) basis functions. The gray nodes together with the white ones are used to define the biquadratic (for quadrangles) and the quadratic (for triangles) basis functions.

Finally, the Element class also inherits, now from the Gauss points class, a set of Gauss points and weights that are written in the local system of coordinates. The Gaussian points are used to perform the numerical integration of the integral terms that appears on the variational formulation of the problem. We used 9 Gauss points to integrate over the quadrangles and 6 points for integrations over a triangle.

Mesh, Dof Handler and FEM Map classes: The Mesh class consists of the ECHE data structure for the domain's mesh discretization. The Mesh class together with the local description of the elements given by the Element class makes possible to create the global indexation of the mesh nodes and its degrees of freedom. The Dof Handler class creates the global indexation and manages its relationship with the local indexation. For example given a local node or degree of freedom and the index of an element, the Dof Handler answers the global index this degree of freedom and vice versa. Figure 6.7 shows an example of the local–global indexation relationship.

In finite element codes, the algebraic computations are performed using a computational domain that is described using a local system of coordinates.

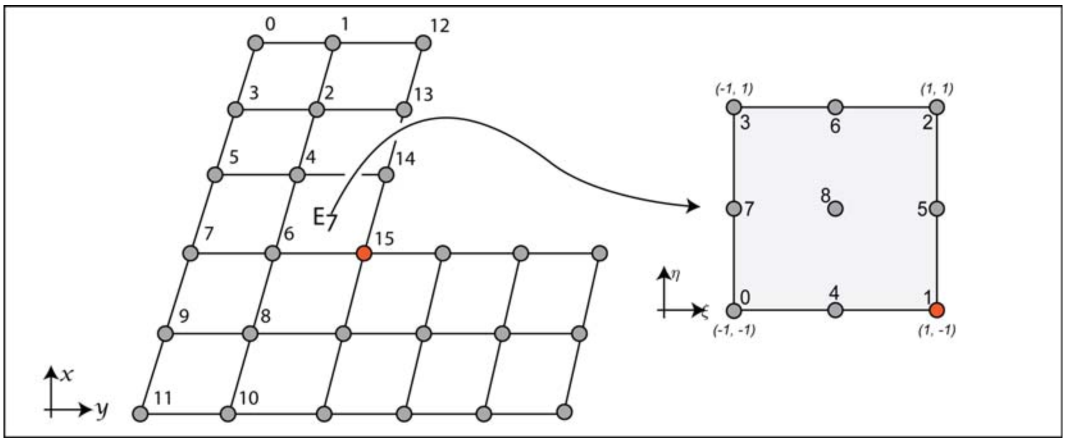

Figure 6.7: Relationship between the local and global indexation. Given an element and the index of a local degree of freedom, with Dof Handler class we can discover its global identifier. In the example of the image, supposing that we have one degree of freedom for each node, the global index of the degree of freedom 1 inside the element 7 is 15.

However, it is necessary to map the local and global coordinate system. The Fem Map class is responsible to perform the computations necessary to perform this change of coordinates system, from the local (ξ, η) to the global (x, y) system.

FEM core class: The central class on our finite element code is the FEM core class. This class manages the main finite element loops: the one in which the non–linear system of differential equations is written as a Jacobian matrix and a residue vector and the main simulation loop.

The assembly loops are performed over the mesh elements. Given an element, the contribution of its nodes to the Jacobian matrix (the residue vector) are computed and later added to the matrix (vector). The Jacobian matrix is a sparse squared matrix which size is $\#dof \times \#dof$ and the residue vector has dimension $\#dof$, where $\#dof$ denotes the number of global degrees of freedom on the mesh. An entry (i, j) on the Jacobian matrix is non zero if and only if the nodes with indexes i and j are both incident to at least one element. The assembly loop for the residue vector is shown in Algorithm 3.

The main simulation loop runs over the time and is responsible to solve Newton's method at each time step. Once Newton's method succeeds, the solution is updated and the simulation continues to the next time step. The

Algorithm 3 Assembly loops

```
for all mesh elements $e$ do
    Le ← ComputeL(e) // Contribution of the nodes inside $e$.
    C ← Assembly(Le) // Puts on the residue vector/Jacobian Matrix.
end for
```

PUC-Rio - Certificação Digital Nº 0610743/CA

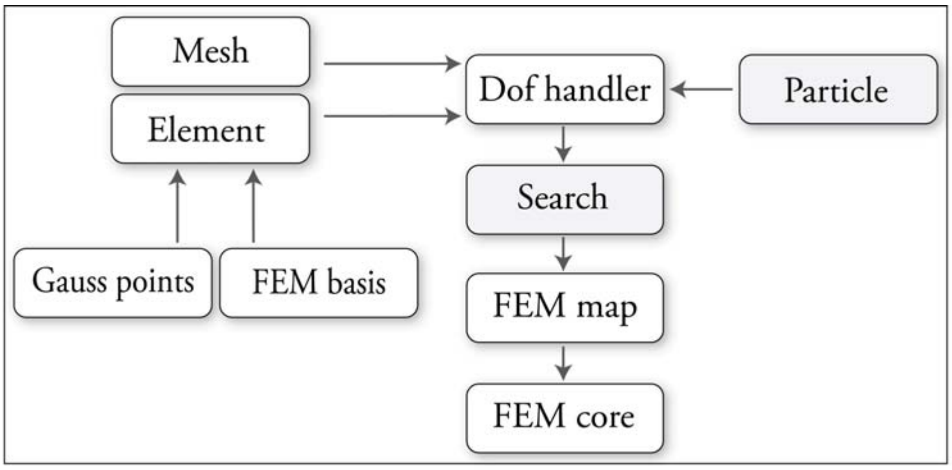

Figure 6.8: An overview of the relation between the most important classes in our fictitious domain code. The differences from the previous implementation are the Particle and Search classes.

loop proceeds until it reaches the maximum time or Newton's method fails.

Observe that the algebraic expressions of the differential equations related to the problem's physics are implemented in this class. If one needs to develop an even more modular code, these computations can be moved to other class that will inherits FEM core elements and will be the responsible for the physical description of the problem.

6.3 Fictitious domain code

The code we developed for fictitious domain simulations of flows with particles is quite similar with the one presented in the previous chapter. The main difference is that we need to include a description for the particles and also to maintain updated a search structure that changes with time. The search structure must answer efficiently when an element or a node is inside any particle, or reciprocally which elements or nodes are inside a given particle. The class relationship on the code is shown in Figure 6.8.

Particle and Search classes: The particle class describes the degrees of freedom of a particle. Given a particle, each of its degrees of freedom is locally indexed, and their association with a globally defined identifier is managed by the Dof handler class. By convention the global indexation of the particle's degrees of freedom begins after the indexation of the flow degrees of freedom.

PUC-Rio - Certificação Digital Nº 0610743/CA

PUC-Rio - Certificação Digital Nº 0610743/CA

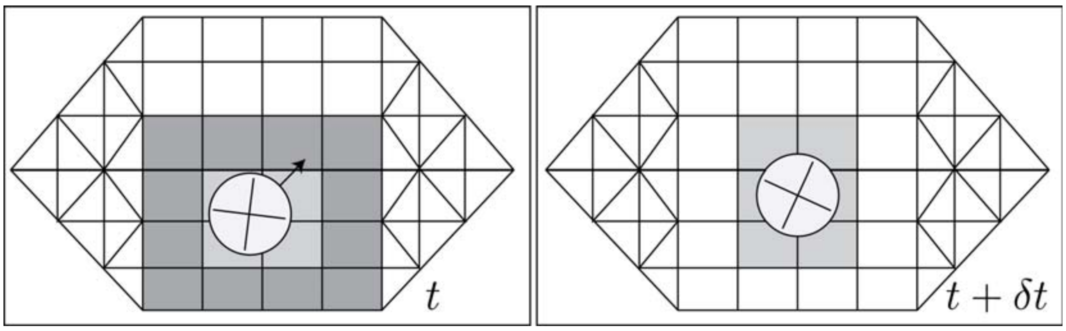

Figure 6.9: Search algorithm: The particle is partially inside the light grey elements and to update the search structure we only need to test the neighbors elements, the dark grey ones in the left image. The update result is shown on the right image.

One of the most important class in our fictitious domain code is the Search class. We use the power of the ECHE data structure and the adjacency structure between elements of the mesh to develop the particle–element search structure.

The search structure uses two integer containers, called Search for Elements and Search for Particles and denoted by SE[] and SP[]. These generic containers stores, respectively, the indexes of elements inside a particle p_i and the indexes of particles inside an element e. The size of SE[] is the number of particles and the size of SP[] is the number of elements on the mesh.

Supposing that the particle displacement is always smaller than the characteristic size of the mesh's elements, we can update efficiently the search structure in each time step, using the ECHE, only testing elements that are inside any particle in the current step, and also its neighboring elements, computed using the data–structure. This makes a huge reduction in the number of tests needed to update the structure and reduces its complexity from $O(n_p \cdot (n_t + n_q))$ to $O(n_p \cdot \bar{n}_e)$, where $\bar{n}_e$ is the average number of elements that are inside or have a neighbor inside a particle.

The update algorithm is sketched in Figure 6.9. The left picture on the image shows an hypothetic search configuration on a given time t. The elements that lie inside the particle are shown in light gray, and together with its neighbors (the dark gray elements) they are the elements that must be tested by the update algorithm. The image at right shows the search structure after the update process.