Chapter 7
The Finite Volume Mesh in OpenFOAM® and uFVM

Abstract The implementation of the finite volume mesh can follow many directions whether in the definition of the mesh fields, the storing of the variables, or even in determining the connectivity relations. This chapter aims at outlining the design decisions that shape the implementation of two CFD codes, uFVM an educational unstructured Finite Volume code and OpenFOAM® an industrial-strength open source code. The two codes are thus presented, initially in terms of their data structure and memory management schemes, and then in terms of how cases are setup. Finally the format of the system of equations generated by each of the two codes are detailed. The reader will notice that while uFVM shares many of the implementation details with OpenFOAM®, its simplicity allows for the use of simpler implementation techniques and data structure.

7.1 uFVM

The unstructured finite volume code uFVM was written to illustrate the various numerical techniques and algorithms, which collectively form a CFD program. Furthermore its numerics are in many ways similar to those in OpenFOAM® [1], making it a good vehicle to understand and present the internals of OpenFOAM®. The main data structures used in uFVM generally mirrors those in OpenFOAM® especially in terms of mesh fields and boundary conditions. Still differences between uFVM and OpenFOAM® will be used to underline various options available to CFD coders and thus to better present some implementation details.

7.1.1 An OpenFOAM® Test Case

The uFVM code is capable of reading an OpenFOAM® mesh included as part of any OpenFOAM® test case. An OpenFOAM® test case is a directory that generally

F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*,
Fluid Mechanics and Its Applications 113, DOI 10.1007/978-3-319-16874-6_7

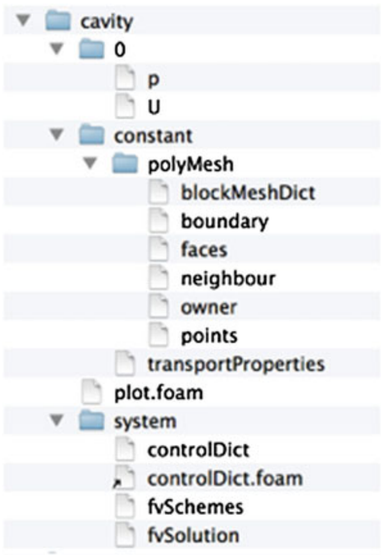


Fig. 7.1 The **cavity** OpenFOAM® case

contains at least three folders. Figure 7.1 shows the innards of the **cavity** test case folder, which consists of the following three sub-folders:

The '**time**' directories contain initialization and boundary condition information about the fields used in the test case. It is also the folder where results for various time steps are stored. The name of each sub-folder refers to the time at which simulation was performed. For example, in the **cavity** tutorial shown in Fig. 7.1, the velocity field **U** and pressure field **p** are initialized from files in the '**0**' folder **0/U** and **0/p**, respectively.

The **system** directory contains at least three files with information about the case setup, the schemes to be used, and the various solution parameters. These files are: (i) **controlDict** which is concerned with the general control parameters of the test case such as the simulation starting and ending times, the time step to be used, and required parameters for data output; (ii) **fvSchemes** in which the discretization schemes used in the simulation are defined; (iii) **fvSolution** that contains information related to the solution algorithms and relaxations used during simulation.

The **constant** directory contains information about relevant physical properties, e.g., **transportProperties**, and the data describing the grid system used within a folder denoted by **polyMesh**.

For the purpose of understanding the finite volume mesh structure in uFVM, attention will be focussed on the **polyMesh** folder in which the information needed to construct the finite volume mesh is defined.

7.1.2 The polyMesh Folder

The polyMesh subdirectory contains the following files:

points
The file **points** is a list of vectors denoting the cell vertices, with vertex **0** being the the first vector in the list, vertex **1** the second vector, etc. The format of the **points** file is shown in Listing 7.1.

```
#number of points
(
(#x #y #z)
…
)
```

Listing 7.1 Storing vertices as vectors of coordinates *x*, *y* and *z*

An example **points** file is shown in Listing 7.2.

```
1074
(
(32 16 0.9377383239)
(33.9429245 16.11834526 0.9377383239)
(35.84160614 16.46798134 0.9377383239)
(37.67648315 17.04080009 0.9377383239)
(39.42799377 17.82870483 0.9377383239)
(41.07658768 18.82359314 0.9377383239)
(…)
…
)
```

Listing 7.2 An example showing how vertices are stored

faces
The file **faces** represents a list of faces, with each face described by a list of indices to vertices in the **points** list, where again, the first entry in the list represents face **0**, the second entry represents face **1**, etc. The format of the **faces** file is shown in Listing 7.3.

```
#number of faces
(
#number of points for face 1 (#p1 #p2 #p3 …)
#number of points for face 2 (#p1 #p2 #p3 … )
…
)
```

Listing 7.3 Storing faces in the form of points of which the face is composed and their indices

Example of a **faces** file is shown in Listing 7.4.

```
3290
(
4(36 573 589 52)
4(41 578 634 97)
4(44 81 618 581)
4(30 82 619 567)
4(121 50 587 658)
4(39 120 657 576)
…
)
```

Listing 7.4 An example showing how faces are stored

owners

The file **owners** is a list in which the owner of faces are stored (Listing 7.5). The position of the owner in the list refers to the face it belongs to. Thus, the owner of face **0** is the index stored in the first entry, the owner of face **1** is the index stored in the second entry, etc. The number of owners is equal to the total number of faces (interior + boundary faces).

The number of Elements is equal to the largest index of the owners.

```
#number of owners
(
#owner of face1
#owner of face2
…
)
```

Listing 7.5 The format used to store owners

An example of an **owners** file is shown in Listing 7.6.

```
3290
(
0
1
2
3
4
5
6
…
)
```

Listing 7.6 An example showing how owners are stored

The total number of cells (**`nCells`**) in the domain can be found in the **owners** file header as shown in Listing 7.7.

```
  note "nPoints:1074  nCells:918  nFaces:3290  nInternalFaces:1300";
```

Listing 7.7 Header of **owners** file

neighbours

A list of neighbor cell labels (Listing 7.8). The number of neighbors is basically equal to the number of interior faces.

```
#number of neighbour
(
#neighbour of face1
#neighbour of face2
…
)
```

Listing 7.8 The format used to store neighbors

An example of a **neighbours** file is shown in Listing 7.9.

```
1300
(
22
68
29
96
31
34
…
)
```

Listing 7.9 An example showing how neighbors are stored

boundary

File **`boundary`** lists the boundaries of the domain, with the faces of each boundary type referred to as a patch and assigned a name. The type of each boundary patch (type) is declared along with its number of faces (nFaces) and the starting face (startFace), which refers to the index of the first face in the list (Listing 7.10).

```
#boundary patch name
{
    type #patchtype;
    nFaces #number of face in patch set;
    startFace #starting face index for patch;
}
```

Listing 7.10 Format of a boundary patch

An example of a wall-type boundary patch is depicted in Listing 7.11.

```
wall-4
{
    type            wall;
    nFaces          100;
    startFace       1300;
}
```

Listing 7.11 An example of a wall-type boundary patch

7.1.3 The uFVM Mesh

In uFVM an OpenFOAM® mesh is read using the **cfdReadOpenFoamMesh** script. The script starts by reading the **points** file into the arrays of struct **nodes** where the (x, y, z) data is stored. Then file **faces** is read and the indices of the nodes are stored into the arrays of struct **faces**. Information about the face patches is then read from the file **boundary**. Finally the files **owners** and **neighbours** are read and the struct **elements** is composed. The loaded data is processed to compute additional geometric and topological information in script **cfdProcessOpenFoamMesh**. The details of a mesh generated over an elbow and read by uFVM is shown in Listing 7.12.

```
m = cfdReadOpenFoamMesh('elbow')
m =
                    nodes: [1x1074 struct]
            numberOfNodes: 1074
            caseDirectory: 'elbow'
            numberOfFaces: 3290
         numberOfElements: 918
                    faces: [1x3290 struct]
    numberOfInteriorFaces: 1300
               boundaries: [1x6 struct]
       numberOfBoundaries: 6
          numberOfPatches: 6
                 elements: [1x918 struct]
        numberOfBElements: 1990
           numberOfBFaces: 1990
```

Listing 7.12 Information uFVM can display about a mesh

As shown in Fig. 7.2 the mesh can be plotted using the cfdPlotMesh command.

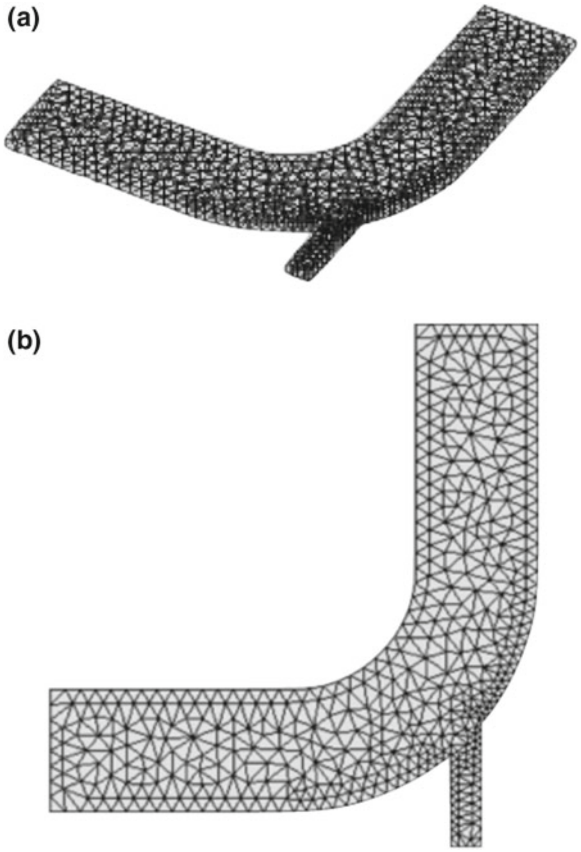


Fig. 7.2 **a** Three dimensional and **b** two dimensional views of the mesh over an elbow displayed using the cfdPlotMesh command

Example of information stored in struct **nodes** is presented in Listing 7.13 by displaying values for the node of index **1**.

```
n1= m.nodes(1)
n1 =
     centroid: [3x1 double]
        index: 1
       iFaces: [172 328 1355 1386 1677 1891 1893]
    iElements: [112 219 220]
```

Listing 7.13 An example of information related to a node

As shown, the centroid contains the coordinates of the node in question, while iFaces and iElements are lists of indices of the faces and elements, respectively, that are connected to the node.

Information stored in struct **faces** can be obtained in a similar way. For example, the attributes of the face indexed **3** are given by (Listing 7.14).

```
m.faces(3)
ans =
                 iNodes: [45 82 619 582]
                  index: 3
                 iOwner: 3
             iNeighbour: 30
               centroid: [3x1 double]
                     Sf: [3x1 double]
                   area: 5.3046
                     CN: [3x1 double]
                geoDiff: 4.5940
                      T: [3x1 double]
                     gf: 0.4226
               walldist: 0
    iOwnerNeighbourCoef: 1
    iNeighbourOwnerCoef: 1
```

Listing 7.14 An example of information related to a face

The above example shows that the struct **faces** contains the list of indices of the nodes defining the face, the indices of the owner and neighbor to the face, the centroid of the face, its surface vector, and its area. In addition, the distance vector **`T`** joining the centroids of the owner and neighbor elements, the geometric factor **`gf`**, the distance vector **`CN`** from the owner element centroid to the surface centroid, and the normal distance to the wall **`walldist`** of the owner element centroid are also stored. Other components will be described later as needed. Moreover, it should be noted that the neighbor for a boundary face is set to **−1**.

As displayed in Listing 7.15 for the element with index **20**, which is a boundary element, the struct **elements** contains three lists of indices for neighboring elements (iNeighbours), faces (iFaces), and nodes (iNodes).

```
m.elements(20)
ans =
                 index: 20
           iNeighbours: [100 103]
                iFaces: [33 34 1317 1493 1494]
                iNodes: [168 79 616 705 617 80]
                volume: 3.2484
              faceSign: [1 1 1 1 1]
    numberOfNeighbours: 2
              centroid: [3x1 double]
```

Listing 7.15 An example of information related to an element

The order of the element indices and faces indices are synchronized such that elements are related to faces in the same order, and boundary faces are defined at the end of the list of faces. The struct **elements** also stores information about the element centroid and its volume. The **faceSign** list indicates whether the element is an owner (+**1**) or neighbor (−**1**) for the respective faces.

Elements can be identified on the mesh as in Fig. 7.3 using the following command (Listing 7.16):

```
cfdPlotElements([20 300])
```

Listing 7.16 Script needed for uFVM to display selected elements [here elements **(20)** and **(300)**] highlighted on the mesh

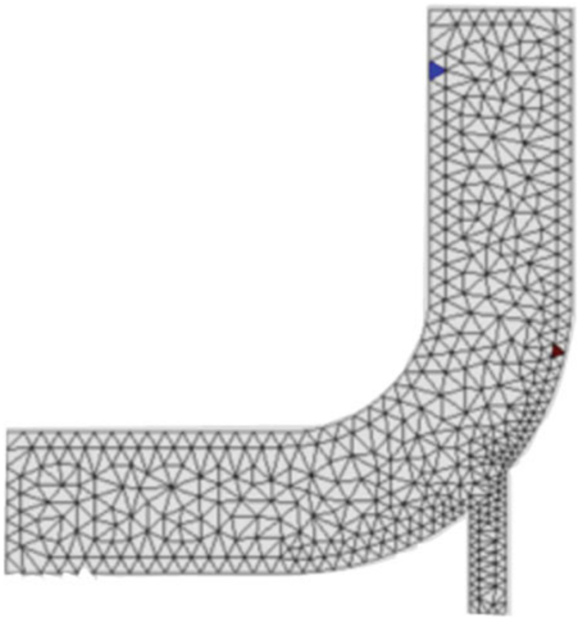


Fig. 7.3 A two dimensional view of the mesh over an elbow highlighting elements **(20)** and **(300)**

Information about element **(20)** were displayed above while the attributes of element **(300)**, which has two boundary faces, are as shown in Listing 7.17.

```
m.elements(300)
ans =
                 index: 300
           iNeighbours: [278 302 590]
                iFaces: [407 435 436 2053 2054]
                iNodes: [283 820 679 142 290 827]
                volume: 1.9083
              faceSign: [-1 1 1 1 1]
    numberOfNeighbours: 3
              centroid: [3x1 double]
```

Listing 7.17 An example of information related to element **(300)** with two boundary faces

Finally information about the various boundary patches, i.e., the list of the boundary faces is stored in struct **boundaries**. Each boundary array contains information about the starting index of the first boundary face, the number of boundary faces belonging to the patch, in addition to the physical type of the patch and its name. For the example considered, information about boundary patch **{1}** is shown in Listing 7.18.

```
>> m. boundaries(1)
ans =
          userName: 'wall-4'
             index: 1
              type: 'wall'
    numberOfBFaces: 100
         startFace: 1301
```

Listing 7.18 An example of information stored for a wall boundary

Moreover, information about the first boundary face, for example, is obtained as shown in Listing 7.19.

```
>> m.faces(1301)
ans =
                 iNodes: [38 53 590 575]
                  index: 1301
                 iOwner: 1
             iNeighbour: -1
               centroid: [3x1 double]
                     Sf: [3x1 double]
                   area: 3.7510
                     CN: [3x1 double]
                geoDiff: 5.6264
                      T: [3x1 double]
                     gf: 1
               walldist: 0.6667
    iOwnerNeighbourCoef: []
    iNeighbourOwnerCoef: []
```

Listing 7.19 Information about a boundary face

To be noted is the index of the first boundary face, which is **1300+1** since in Matlab® arrays start at index **1** while in the C computer language arrays start at index **0**.

Thus to loop over the faces of a certain patch, the index of the starting face and the number of faces from the struct **boundary** associated with the said patch are needed. For example to loop over the faces of patch 2, the following script (Listing 7.20) can be used:

```
theMesh = cfdGetMesh;
iPatch = 2;
iBFaces = cfdGetFaceIndicesForBoundaryIndex(iPatch)
for iBFace=iBFaces
      theBFace = theMesh.faces(iBFace);
      disp(theBFace) %display theBFace internal fields
end
```

Listing 7.20 Looping over boundary patch faces

cfdGetFaceIndicesForBoundaryIndex is defined in Listing 7.21 as follows:

```
theIndices = cfdGetFaceIndicesForBoundaryIndex(theBoundaryIndex)
%
theBoundary = cfdGetBoundary(theBoundaryIndex);
theNumberOfBFaces = theBoundary.numberOfBFaces;
theStartFace = theBoundary.startFace;
theIndices = [theStartFace:theStartFace+theNumberOfBFaces-1];
%
end
```

Listing 7.21 Indices of boundary faces for a specific patch

7.1.4 uFVM Geometric Fields

In addition to all data stored in the struct **mesh**, information about the model to be solved and the values of the fields of interest should be stored to be accessed when needed. Three types of locale can be identified and fields of different types can be defined on them. Namely for locale (Elements, Faces, and Nodes) and for types (scalars, vectors, and tensors). The various fields are first defined based on their locale.

7.1.4.1 The Element Fields

An element field is constructed using the following script (Listing 7.22):

```
cfdSetupMeshField(theUserName,theLocale,theType,theTimeStep)
```

Listing 7.22 Script needed to construct an element field

where **theUserName** is the name of the field, **theLocale** is the geometric entity over which it is defined **(Elements**, **Faces**, **Nodes)**, **theType** defines the type of the array elements (**Scalar** or **Vector**), and finally the **TimeStep** indicates the relative time of the field (Step0, Step1, etc.). For example, the following script (Listing 7.23):

```
>> UField = cfdSetupMeshField('U:water','Elements','Vector','Step0')
UField =
    userName: 'U:water'
        name: 'U_fluid01'
        type: 'Vector'
      locale: 'Elements'
         phi: [2908x3 double]
```

Listing 7.23 Example of setting up a vector field

sets up a vector field defined over **Elements** at time step 0, i.e., at the current time step.

As shown in Fig. 7.4, the array has the size of the **NumberOfElements+ theNumberOfBoundaryFaces** since the element array will include the values defined for each element in the mesh in addition to each boundary face. The boundary face values represent boundary conditions for that field. These boundary values are grouped in terms of boundary patches.

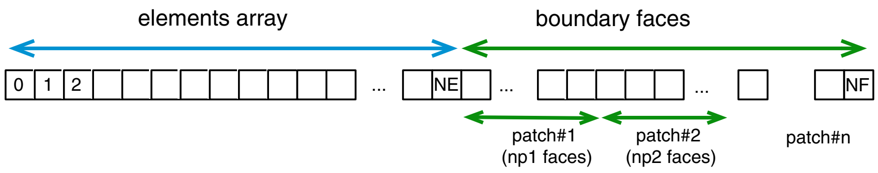


Fig. 7.4 Size of the field array

Generally they can be accessed as follows: For example to initialize the boundary values of the UField at patch **1** to a value **[1 0 0]**, the following should be written (Listing 7.24):

```
% get the mesh
theMesh = cfdGetMesh;
% get information about the boundary patch
theBoundary = theMesh.boundaries(iPatch);
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;
numberOfBFaces = theBoundary.numberOfBFaces;
% Starting face
iFaceStart = theBoundary.startFace;
% get information about starting and ending elements
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
% define the indices as an index array
iBElements = iElementStart:iElementEnd;
>> UField.phi(iBElements,:) =
cfdComputeFormulaAtLocale('[1;0;0]','BPatch1','Vector')
ans =
     1     0     0
     1     0     0
     1     0     0
…
       …
```

Listing 7.24 The script needed to initialize the boundary value of a field in a patch

where the statement in Listing 7.25 given by

```
cfdComputeFormulaAtLocale(theFormula,theLocale,theType)
```

Listing 7.25 Statement used to compute values over a locale

evaluates the values in **theFormula** over **theLocale** and returns and array of the appropriate length of type **theType** (**Scalar** or **Vector**).

7.1.4.2 The Face Fields

A face field is constructed with **theLocale** set to '**Faces**' as (Listing 7.26)

```
cfdSetupMeshField(theUserName,theLocale,theType,theTimeStep)
```

Listing 7.26 Statement used to construct a face field

Again, as shown in Fig. 7.5, the length of the array is equal to **numberOfFaces**, which combines the **numberOfInteriorFaces** plus the sum of all boundary faces.

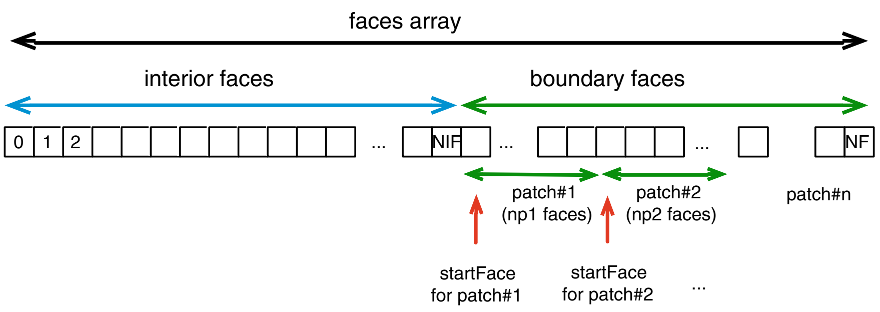


Fig. 7.5 Size of **faces** array

The boundary **faces** for any boundary patch can be accessed as (Listing 7.27)

```
theMesh = cfdGetMesh;
numberOfElements = theMesh.numberOfElements;
numberOfInteriorFaces = theMesh.numberOfInteriorFaces;

theBoundary = theMesh.boundaries(iPatch);
numberOfBFaces = theBoundary.numberOfBFaces;

%
iFaceStart = theBoundary.startFace;
iFaceEnd = iFaceStart+numberOfBFaces-1;
iBFaces = iFaceStart:iFaceEnd;
%
iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
iElementEnd = iElementStart+numberOfBFaces-1;
iBElements = iElementStart:iElementEnd;

thBFaces = theMesh.faces(iBFaces)
```

Listing 7.27 Script to access the various arrays of struct **faces**

and the boundary elements for a scalar field can then be retrieved using Listing 7.28 as

```
phi_b = phi[iBElements]
```

Listing 7.28 Accessing the boundary elements of a scalar phi

7.1.4.3 The Node Field

The node field is a list where the indices of vertices of faces are stored. Each face is referred to by the indices of its vertices in the **points** list (Fig. 7.6). The position of the face in the list refers to the face index. Therefore face **0** is the first entry in the list, face **1** is the second entry in the list, and so on.

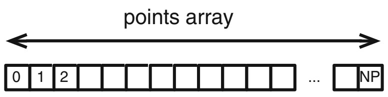


Fig. 7.6 A **points** array

7.1.5 *Working with the uFVM Mesh*

Looping over elements, interior faces, boundary faces, boundary elements, or even boundary patches are common operations that are performed during the discretization and solution cycles. It is worth reviewing the mechanisms that allow performing these operations readily.

7.1.5.1 Looping Over Elements

This is a simple loop to implement as the number of elements is already known and the elements are indexed from **1** to the **numberOfElements** (**0** to **numberOfElements-1** for OpenFOAM®). Also element fields are indexed in a similar fashion, so accessing them is as simple as accessing elements. Thus the loop is simply as shown in Listing 7.29.

```
for iElement=1:numberOfElements
      theElement = theMesh.elements(iElement)
      phi(iElement)  % this is field phi at centroid of element iElement
      ....
end
```

Listing 7.29 Script used to loop over elements

To access the boundary elements, the script shown in Listing 7.30 is used.

```
for boundary = 1:numberOfBoundaries
    theBoundary = theMesh.boundaries{1};
end
```

Listing 7.30 Script used to access boundary elements

7.1.5.2 Looping Over Faces

The faces array is constructed so that all interior faces run through indices **1** to **numberOfInteriorFaces** followed by the boundary faces that are also arranged according to the boundary patch to which they belong. Thus looping over the interior faces is straight forward and is written as in Listing 7.31.

```
for iFace=1:numberOfInteriorFaces
      theFace = theMesh.faces(iFace)
end
```

Listing 7.31 Script used to loop over interior faces

Looping over the boundary faces can be done as shown in Listing 7.32.

```
for iBFace= numberOfInteriorFaces+1:numberOfFaces
      theBFace = theMesh.faces(iBFace)


end
```

Listing 7.32 Script used to loop over boundary faces

If the interest is to loop over the boundary faces of a certain patch, then the loop runs from a **start** face defined in the boundary condition to **nFaces** also defined in the boundary condition. Thus a loop over the faces of boundary patch **n** would be written as depicted in Listing 7.33.

```
startFace = theMesh.boundaries(n).startFace
nFaces = theMesh.boundaries(n).nFaces
for iBFace = startFace: startFace+nFaces-1
```

Listing 7.33 Script used to loop over boundary faces of a certain patch

The subroutine **cfdGetFaceIndicesForBoundaryIndex** is used to directly get the vector **startFace: startFace+nFaces-1**.

7.1.6 Computing the Gauss Gradient

Computing the gradient of an element field in uFVM, necessitates the use of many of the above routines. Function **cfdComputeGradientGauss0**, displayed in Listing 7.34, shows the details of the Gauss Gradient implementation.

```
function phiGrad = cfdComputeGradientGauss0(phi,theMesh)
%====================================================
%  written by the CFD Group @ AUB, Fall 2014
%====================================================
%
if(nargin<2)
    theMesh = cfdGetMesh;
end
%-----------------------------------------------------
% Initialize phiGrad Array
%-----------------------------------------------------
phiGrad = zeros(theMesh.numberOfElements+theMesh.numberOfBFaces,3);
%-----------------------------------------------------
% INTERIOR FACES contribution to gradient
%-----------------------------------------------------
% get the list of interior faces indices and the list of boundary faces
indices
iFaces = 1:theMesh.numberOfInteriorFaces;
iBFaces = theMesh.numberOfInteriorFaces+1:theMesh.numberOfFaces;
% get the list of element indices and the list of boundary element indices
iElements = 1:theMesh.numberOfElements;
iBElements = theMesh.numberOfElements+1:theMesh.numberOfElements
+theMesh.numberOfBFaces;
% get the list of owners and neighbours for all interior faces
iOwners = [theMesh.faces(iFaces).iOwner]';
iNeighbours = [theMesh.faces(iFaces).iNeighbour]';
% get the surface vector of all interior faces
Sf = [theMesh.faces(ifaces).Sf]’;

% get the geometric factor for all interior faces
gf = [theMesh.faces(iFaces).gf]';
% compute the linear interpolation of phi into all interior faces
phi_f = gf.*phi(iNeighbours,iComponent) + (1-gf).*phi(iOwners,iComponent);
% loop over faces and add contribution of phi face flux to the owner and
neighbour elements of the faces
for iFace=iFaces
    phiGrad(iOwners(iFace),:) = phiGrad(iOwners(iFace),:) +
                                 phi_f(iFace)*Sf(iFace,:);
    phiGrad(iNeighbours(iFace),:) = phiGrad(iNeighbours(iFace),:)
                                 phi_f(iFace)*Sf(iFace,:);
    end
%-----------------------------------------------------
% BOUNDARY FACES contribution to gradient
%-----------------------------------------------------% get the list of
elements owning a boundary face
iOwners_b = [theMesh.faces(ibFaces).iOwner]’;
% get the boundary values of phi
phi_b = phi(iBElements,iComponent);
% get the surface vector of all boundary faces
Sb = [theMesh.faces(iBFaces).Sf]';
% loop over all boundary faces and add phi flux contribution to gradient
for k=1:theMesh.numberOfBFaces
    phiGrad(iOwners_b(k),:) = phiGrad(iOwners_b(k),:) + phi_b(k)*Sb(k,:);
end
```

Listing 7.34 Computing the Gauss gradient

```
%-----------------------------------------------------
% Get Average Gradient by dividing with element volume
%—————————————————————————————
% get volume of all elements in mesh
volumes = [theMesh.elements(iElements).volume]’;
% compute the average gradient by dividing the sum of all phi fluxes for an
element by the volume of the element
for iElement =1:theMesh.numberOfElements
    phiGrad(iElement,:) = phiGrad(iElement,:)/volumes(iElement);
end
%-----------------------------------------------------
% Set boundary Gradient equal to associated element Gradient
%—————————————————————————————
phiGrad(iBElements,:) = phiGrad(iOwners_b,:);
```

Listing 7.34 (continued)

The loop over the boundary faces in Listing 7.34 did not take into account the patches to which the boundary faces belong, rather it just looped over all boundary faces. A version that would loop over the various patches and then over their respective boundary faces would be written as shown in Listing 7.35.

```
%-----------------------------------------------------
% BOUNDARY FACES contribution to gradient
%-----------------------------------------------------
% get the list number of boundary Patches
%
theNumberOfPatches = theMesh.numberOfBoundaries;
theEquation = cfdGetModel(theEquationName);
%

for iPatch=1:theNumberOfPatches
    %
    theBoundary = theMesh.boundaries(iPatch);
    numberOfBFaces = theBoundary.numberOfBFaces;
    %
    iFaceStart = theBoundary.startFace;
    iFaceEnd = iFaceStart+numberOfBFaces-1;
    iBFaces = iFaceStart:iFaceEnd;
    %
    iElementStart = numberOfElements+iFaceStart-numberOfInteriorFaces;
    iElementEnd = iElementStart+numberOfBFaces-1;
    iBElements = iElementStart:iElementEnd;

    iBOwners = [theMesh.faces(iBFaces).iOwner]?;

    phi_b = phi(iBElements,iComponent);
    Sb = [theMesh.faces(iBFaces).Sf]';
    % loop over all boundary faces and add phi flux contribution to gradient
    for k= 1: numberOfBFaces
        phiGrad(iBOwners(k),:) = phiGrad(iBOwners(k),:) + phi_b(k)*Sb(k,:);

    end
end
```

Listing 7.35 Boundary faces contribution to gradient implemented by looping over the various patches and then over their respective boundary faces

7.2 OpenFOAM®

OpenFOAM® [1] uses a finite volume cell-centered discretization of the domain and handles unstructured mesh data format based on the so called face-addressing storage. The aim of this data structure is to provide maximum flexibility in the definition of unstructured grids in order to allow for the use of a general polyhedron shape.

As shown in Fig. 7.7, a polyhedron is a three-dimensional solid consisting of a collection of plane polygons, joined at their edges. Tetrahedra (polyhedra with four triangular faces) and Hexahedra (polyhedra with six faces), for instance, are polyhedra where the faces are made by polygons of three and four edges, respectively. The possibility to describe any three-dimensional shape and use it as a finite volume element for the discretization of the equations gives multiple advantages and flexibility during mesh generation.

An efficient way to describe a mesh constructed using polyhedral elements is the face addressing. In face addressing the element shape is of no consequence to the equation discretization process, requiring the use of global addressing when forming the coefficients. (Details on the formation of coefficients from the local and global perspective will be detailed later on).

In OpenFOAM® the data for points, faces, and elements are stored in a number of lists (arrays), as shown in Fig. 7.8. The list of points contains the three dimensional spatial coordinates, defined as vectors, which correspond to the vertices of the actual mesh. The dimensional unit is the meter. Moreover each vertex has a label defined by the position in the list, and because of the use of the C++ computer language the label counting starts from zero.

The faces are defined by a list of vertex labels referring to points in which the ordering is such that each two adjacent points are connected by an edge. The face list is organized in a way that all internal faces appear first in the list followed by faces related to the first boundary, then faces related to the second boundary, and so on. Lists of boundary faces are also named *patches*. It is important to remember that internal faces belong to two cells while boundary faces belong to one cell only.

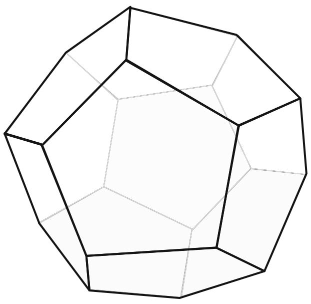


Fig. 7.7 A polyhedron element

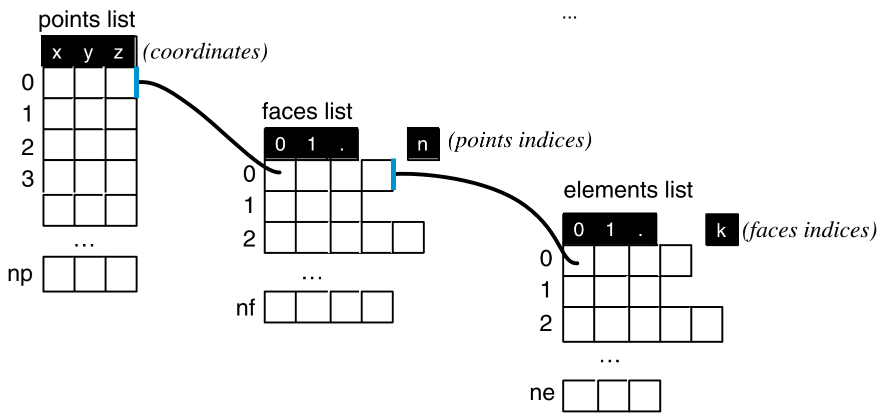


Fig. 7.8 Points, faces, and elements lists in OpenFOAM®

Finally the element or cell list is defined by a list of indices, where the position in that list is the cell index, the first index at any position is the number of faces for that element, and the face indices at a given position represent the faces for that cell.

OpenFOAM® can read computational grids from any mesh generator software capable of writing the mandatory files needed by OpenFOAM®. As explained earlier, the mesh is named **polyMesh** and has to be defined with a set of proper files. These files are placed in the directory "*constant/polyMesh*". Based on the previous description the names of the files whose contents are self-explanatory are given by

- **points**
- **faces**
- **owner**
- **neighbour**
- **boundary**

In OpenFOAM® the entire boundary of the domain is described as a list of patches in the file **boundary**. The file **boundary** is a list of all defined patches containing a proper entry named **dictionary** with each patch declared using the patch name, its type, its number of faces (**nFaces**), and the starting face (**startFace**). The general syntax used is as written in Listing 7.36.

```
PatchName
{
    type patch;
    nFaces 'number of faces';
    startFace 'starting face';
}
```

Listing 7.36 Script used to create a patch

An example of a **boundary** file with two boundary types is shown in Listing 7.37.

```
2
(
    inlet
    {
        type            patch;
        physicalType    automatic;
        nFaces          100;
        startFace       9850;
    }
    outlet
    {
        type            patch;
        physicalType    automatic;
        nFaces          100;
        startFace       9950;
    }
)
```

Listing 7.37 A **boundary** file with two boundary types

From a programming point of view it is worth giving a brief introduction of the C++ classes that handle the mesh and allow access to specific data.

The base class that handles the "low-level" structure is called **primitiveMesh**. It is a generic class that wraps the geometric information of the mesh without assuming any particular form of discretization. It is the basic class for low level information about the mesh.

Examples of some of the functions that are members of the **primitiveMesh** class are shown in Listing 7.38.

```
const labelListList &      cellCells() const
const labelListList &      pointCells() const
const cellList &           cells() const
const vectorField &        cellCentres() const
const vectorField &        faceCentres() const
const scalarField &        cellVolumes() const
const vectorField &        faceAreas() const
```

Listing 7.38 Some functions of the **primitiveMesh** class

As can be seen, this class allows obtaining specific data of the mesh (e.g., cell volume, cell centers, face centers, etc.). It is worth noting that the class itself does not recognize boundaries or domain interfaces. Boundaries and domain interfaces are defined in the **polyMesh** class, which is derived from the **primitiveMesh** class. In addition to possessing all the attributes of the derived class, the **polyMesh** class introduces the handling of boundary definition and information, as shown in Listing 7.39. Again this class does not require any particular discretization scheme.

```
const labelUList &  owner () const   //Internal face owner.
const labelUList &  neighbour () const   //Internal face neighbour.
const DimensionedField< scalar, volMesh > &    V0 () const
      Return old-time cell volumes.
const DimensionedField< scalar, volMesh > &    V00 () const
      Return old-old-time cell volumes.
 tmp< surfaceVectorField >       delta () const
      Return face deltas as surfaceVectorField
```

Listing 7.39 Information that can be obtained from the **polymesh** class

While **primitiveMesh** and **polyMesh** are the basic classes of the polyhedral mesh, the **fvMesh** class is derived from **polyMesh** and adds the particular data and functions needed for the finite-volume discretization. Addressing information as well as boundary information and specific mesh data are accessible for finite volume discretization.

OpenFOAM® decomposes the boundary mesh into patches stored in a list designated by **polyPatchList** under the class **polyBoundaryMesh**. Then, similar to interior mesh, a specialized class denoted by **fvBoundaryMesh** is derived from **polyBoundaryMesh** that inherits its functionalities and expands on it to include specific functions and data needed for finite-volume discretization. As for internal discretization, a similar hierarchy of classes composed of **primitivePatch**, **polyPatch**, and **fvPatch** is defined for boundary discretization. These classes are specific for the boundary mesh and contain the geometric information of each boundary. But it is the **fvPatch** that is used to implement the boundary conditions during the finite volume discretization. Figure 7.9 presents a schematic of the basic mesh description used in OpenFOAM®.

In a similar way the **fvMesh** is used to access all mesh functionalities. Thus, it is mainly with **fvMesh** and **fvPatch** that the discretization classes and functions interact.

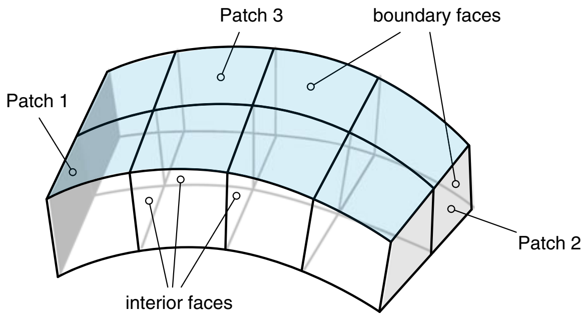


Fig. 7.9 Schematic of the basic mesh description used in OpenFOAM® [1]

Examples of codes for reading and accessing the main properties of a mesh within the OpenFOAM® framework are now presented.

To read a mesh, the **fvMesh** class that handles the finite volume mesh and discretization is needed along with a special constructor as shown in Listing 7.40.

```
#include "setRootCase.H"
#include "createTime.H"
Foam::Info
    << "Create mesh for time = "
    << runTime.timeName() << Foam::nl << Foam::endl;

Foam::fvMesh mesh
(
    Foam::IOobject
    (
        Foam::fvMesh::defaultRegion,
        runTime.timeName(),
        runTime,
        Foam::IOobject::MUST_READ
    )
);
```

Listing 7.40 Script to read a mesh in OpenFOAM®

The *include* statements in Listing 7.40 are required for initialization before reading the mesh files.

Once the **fvMesh** instantiation is constructed under the variable named *mesh*, the manipulation of the loaded mesh and collection of the necessary data for further programming can proceed, as shown in Listing 7.41.

```
    // read the element centroids
    volVectorField C = mesh.C();
    // element volumes
    volScalarField V = mesh.V();
    // surface centroids
    surfaceVectorField Cf = mesh.Cf();
```

Listing 7.41 Extracting data after reading a mesh in OpenFOAM®

To loop over the cell volumes, the dedicated class function for information on volumes should first be called (Listing 7.42). This is accomplished via

```
const DimensionedField< scalar, volMesh >&cellVolumes = mesh.V();
```

Listing 7.42 Getting information on volumes

and then a loop over the cell volumes can be performed as shown in Listing 7.43.

```
    forAll(cellVolumes, cellI)
    {
       scalar cellVolume = cellVolumes[cellI];
    }
```

Listing 7.43 Looping over all cell volumes

In OpenFOAM® the standard "for" loop is replaced by a more compact syntax using a macro definition named **forAll** (Listing 7.44) defined in the UList.H file as

```
#define forAll(list, i)   \
        for (Foam::label i=0; i<(list).size(); i++)
```

Listing 7.44 Script used to replace the standard "for" loop by the macro **forAll**

In a similar way access to other mesh information can be performed as shown in Listing 7.45.

```
    // internal access
    forAll(cellCenters.internalField(), cellI)
    {
       vector cellCenter = cellCenters[cellI];
    }
    // internal access
    forAll(faceNormals.internalField(), faceI)
    {
       vector faceNormal = faceNormals.internalField()[faceI];
    }
]
```

Listing 7.45 Script used to access mesh information

For boundary patches the access procedure is similar except that now each patch has its own class definition, and the full list of patches is defined in the **fvBoundaryMesh** class containing the **fvPatches** list. Therefore to access boundary data the following is used (Listing 7.46):

```
    const fvBoundaryMesh& boundaryMesh = mesh.boundary();
    forAll(boundaryMesh, patchI)
    {
        const fvPatch& patch = boundaryMesh[patchI];
        forAll(patch, faceI)
        {
            vector faceNormal = patch.Sf()[faceI];
            scalar faceArea = patch.magSf()[faceI];
            vector unitFaceNormal = patch.nf()()[faceI];
            vector faceCenter = patch.Cf()[faceI];
            label owner = patch.faceCells()[faceI];
        }
    }
```

Listing 7.46 Accessing data on boundary patches

7.2.1 Fields and Memory

Within the OpenFOAM® generic framework, lists, arrays, and in general containers of different types and sizes can be defined. For a given mesh and computational structure, it would be useful to define a specific class capable of combining fields, lists, and vectors directly with the mesh. A useful class that satisfies this requirement is the template class **GeometricField**<*Type*,…>. Each data defined using this class is strictly related to the mesh dimensions, both for the boundaries and interior domain (be it the number of elements, the number of interior faces or even the number of interior vertices).

In general the template class **GeometricField**<*Type*,…> stores the following data structure based on three main different characteristics of the mesh:

- **volField**<*Type*> A field defined at cell centers;
- **surfaceField**<*Type*> A field defined on cell faces;
- **pointField**<*Type*> A field defined on cell vertices.

The class also inherits the following properties:

- **Dimensions**: OpenFOAM® provides a useful feature in handling fields under the class **GeometricField**. OpenFOAM® associates with each field a dimension (meter, kg, seconds, etc.) characterizing the physical meaning of the variable. Based on that all operations carried out with **GeometricField** have to be performed with fields of the same dimension (e.g., velocity can be summed up with velocity not pressure) otherwise a runtime error transpires during execution. Moreover the dimension of any new field defined as a combination of existing fields in **GeometricField**, is automatically generated by OpenFOAM® through applying the same algebraic relation utilized to generate the new field on the dimensions of the used fields (e.g., dividing a mass field with a volume field results in a field having the dimension of density).
- **InternalField**: This is a repository of size equal to the size of the internal mesh properties (cell centers, vertices, or faces) in which internal information of a defined field is stored.
- **BoundaryField**: contains all information pertinent to a defined variable at the boundary. A list of patches is developed and each field is defined for the patches of the boundary with the name **GeometricBoundaryField**. Operations can be performed either on the full set of boundaries or on a specific patch using **fvPatchField**.
- **Mesh**: Being a class strictly related to the mesh, each **GeometricField** contains a reference to the corresponding mesh.
- **Time Values and Previous Values**: This class is required to handle the specific field during simulation. It stores information of the previous two time steps for second order accuracy in time.

In the following, examples on the use of the specific class **GeometricField** to access the main properties of a field associated with a mesh are provided.

In the first example it is required to define the two variables U and T, representing a velocity and a temperature field, respectively, at the cell center of the mesh. This will be done using the templates of the specialized class **GeometricField**, which supports scalars, vectors, and tensors data type. The script used is shown in Listing 7.47 in which one of the several constructors of the class **GeometricField** *is used* (other constructors can be found in the header file of the specific class *GeometricField.H*).

```
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("DTVol", dimensionSet(0,0,0,1,0,0,0), 300.0),
    "zeroGradient"
 );

volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedVector("U",dimensionSet(0,1,-1,0,0,0,0),vector::zero),
    "zeroGradient"
);
```

Listing 7.47 Script used to define the two variables U and T in OpenFOAM®

As can be seen in the code, both fields are linked to the *mesh*. Both require specifying the following *four arguments*: (i) field name, (ii) field dimension, (iii) initialization, and (iv) boundary conditions. Specific boundary conditions have to be defined for the field and in this case a zero order extrapolation (i.e., “zeroGradient”) is specified for the full set of boundaries and for both variables.

Similar constructors can be used for variables defined at faces of the mesh. For example, the script needed to define the mass flux (volume flow rate) field at cell faces is shown in Listing 7.48.

```
    surfaceScalarField mdot
    (
        IOobject
        (
            "mdot",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        linearInterpolate(U) & blockMesh.Sf()
    );
```

Listing 7.48 Script used to define the mass flux field at mesh faces

In this case the field is constructed with a scalar product between velocity (interpolated at the face) and face area vectors. This field represents the volume flow rate at control volume faces. It also represents the mass flow rate field for incompressible flows with a density value of one.

Once fields are defined, access to specific data in different parts of the mesh is possible as shown in the scripts below.

7.2.2 *InternalField Data*

Internal field data can be accessed (Listing 7.49) using

```
// internal access
forAll(T.internalField(), cellI)
{
   scalar cellT = T.internalField()[cellI];
}
// internal access
forAll(U.internalField(), cellI)
{
   vector cellU = U.internalField()[cellI];
}
```

Listing 7.49 Accessing internal field data

7.2.3 *BoundaryField Data*

Boundary field data can be accessed (Listing 7.50) using

```
const volVectorField::GeometricBoundaryField& UBoundaryList =
U.boundaryField();
// boundary access
forAll(UBoundaryList, patchI)
{
    const fvPatchField<vector>& fieldBoundary = UBoundaryList[patchI];
    forAll(fieldBoundary, faceI)
    {
        vector faceU = fieldBoundary[faceI];
    }
}
```

Listing 7.50 Accessing boundary field data

or in a more compact form via (Listing 7.51)

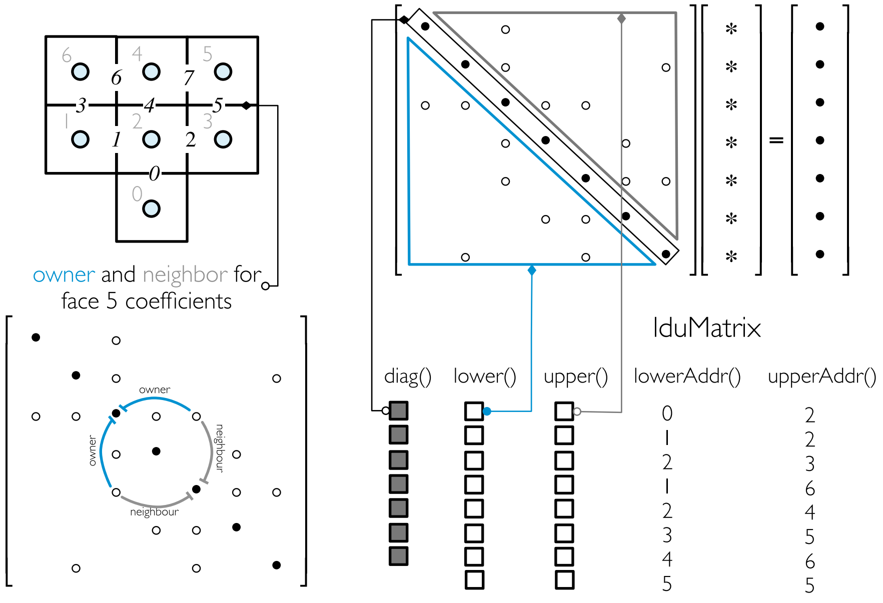
```
// boundary access
forAll(T.boundaryField(), patchI)
{
    forAll(T.boundaryField()[patchI], faceI)
    {
        scalar faceT = T.boundaryField()[patchI][faceI];
    }
}
```

Listing 7.51 A more compact script to access boundary field data

7.2.4 *lduAddressing*

OpenFOAM® uses exclusively **face addressing** in its discretization loops and coefficients storage. It also uses arbitrary polyhedral elements in its meshes. A polyhedron element can have any number of faces with a neighbor element associated with each interior face. The storage of coefficients in OpenFOAM® is based on the **face addressing scheme**. In this approach, coefficients are stored following the interior face ordering, with access to elements and their coefficients, based on the owner/neighbor indices associated with interior faces. As mentioned in the previous section, the element with the lower index is the **owner** while the **neighbor** is the element with the higher index. For boundary faces the owner is always the cell to which the face is attached while no neighbor is defined by setting the **neighbour** index to $-\mathbf{1}$. The list of **owner** or **neighbor** indices thus define the order in which the element-to-element coefficients are assembled for the various integral operators.

The above described scheme is denoted by **lduAddressing** and is implemented in the lduMatrix class displayed in Fig. 7.10 that includes 5 arrays representing the diagonal, upper, and lower coefficients and the lower and upper indices of the face owner and neighbor, respectively. In this scheme, owners represent the lower triangular part of the matrix (lower addressing) while neighbors refer to the upper triangular part (upper addressing). It is worth mentioning that given a face for an owner element the lower and upper addressing provide respectively the column and row where the coefficient of the face flux is stored in the matrix while it is the opposite for a neighbor cell. For the domain shown on the upper left side of Fig. 7.10, the owner and neighbor of each face are displayed in the **lowerAddr()** and **upperAddr()** array, respectively. The face number is the position of the owner or neighbor in the array plus one (since numbering starts with 0). Considering internal face number 4, for example, its related information is stored in the fifth row (in C++ indexing starts from 0) of the **lower()**, **upper()**, **lowerAddr()**, and **upperAddr()** array, respectively. The stored data indicate that its owner is element number 2 **(lowerAddr()** array), its neighbor is element number 4 **(upperAddr()** array), the coefficient multiplying ϕ_4 in the algebraic equation for element number 2



lowerAddr()	upperAddr()
0	2
1	2
2	3
1	6
2	4
3	5
4	6
5	5

Fig. 7.10 The lduAddressing and lduMatrix

is stored in the fifth row of the array **upper()**, and the coefficient multiplying ϕ_2 in the algebraic equation for element number 4 is stored in the fifth row of the array **lower()**.

Thus the lduAdressing provides information about the addresses of the off diagonal coefficients in relation to the faces to which they are related. This means that while the computational efficiency for various operations on the matrix is high when they are mainly based on loops over all faces of the mesh, direct access to a specific row-column matrix element is difficult and inefficient. One example is the summation of the off-diagonal coefficients for each row given by

$$a_c = \sum_{n \sim nb(C)} a_n \tag{7.1}$$

In this case the use of face addressing does not allow direct looping over the off-diagonal elements of each row and performing such operation requires looping over all elements because it can only be done, as shown in Listing 7.52, following a face based approach.

```
for (label faceI=0; faceI<l.size(); faceI++)
{
    ac[l[faceI]] -= Lower[faceI];
    ac[u[faceI]] -= Upper[faceI];
}
```

Listing 7.52 Summation of the off-diagonal coefficients

In Listing 7.52 ac is the sum of the off-diagonals coefficients, *l* and *u* are the upper and lower addressing while *Lower* and *Upper* stores the corresponding coefficients. As can be noticed the summation is performed looping through all the faces and the summation of each row is not sequential, depending only on the owner-neighbor numeration of the mesh.

So in general the lduAddressing introduces a more complex handling of the matrix operations and it will be reflected also in the implementation of linear solvers but with the advantage of higher computational speed.

7.2.5 Computing the Gradient

The script used to compute the Green-Gauss gradient in OpenFOAM® is shown in Listing 7.53.

```
Foam::fv::gaussGrad<Type>::gradf
(
    const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf,
    const word& name
)
{
    typedef typename outerProduct<vector, Type>::type GradType;
    const fvMesh& mesh = ssf.mesh();
    tmp<GeometricField<GradType, fvPatchField, volMesh> > tgGrad
    (
        new GeometricField<GradType, fvPatchField, volMesh>
        (
            IOobject
            (
                name,
                ssf.instance(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensioned<GradType>
            (
                "0",
                ssf.dimensions()/dimLength,
                pTraits<GradType>::zero
            ),
            zeroGradientFvPatchField<GradType>::typeName
        )
    );
    GeometricField<GradType, fvPatchField, volMesh>& gGrad = tgGrad();
    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();
    const vectorField& Sf = mesh.Sf();
//
    Field<GradType>& igGrad = gGrad;
    const Field<Type>& issf = ssf;
    forAll(owner, facei)
    {
        GradType Sfssf = Sf[facei]*issf[facei];

        igGrad[owner[facei]] += Sfssf;
        igGrad[neighbour[facei]] -= Sfssf;

    }
    forAll(mesh.boundary(), patchi)
    {
        const labelUList& pFaceCells =
            mesh.boundary()[patchi].faceCells();

        const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
        const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];
        forAll(mesh.boundary()[patchi], facei)
        {
            igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
        }
    }
    igGrad /= mesh.V();
    gGrad.correctBoundaryConditions();
    return tgGrad;
}
```

Listing 7.53 Script used to compute a gradient field in OpenFOAM®

The small introduction reported above is neither intended as a replacement of OpenFOAM® user's manual [1] nor a replacement of a C++ manual. The purpose of the above presentation is to introduce the reader to the philosophy and general concepts that are useful for a quick understanding of the OpenFOAM® framework. Despite the generality of presentation, the described syntax introduced one of the main important data defined within OpenFOAM® that are necessary for writing a complete solver, which will be described in later chapters.

7.3 Mesh Conversion Tools

There are many tools capable of converting mesh files from a variety of format to the OpenFOAM® format. Some of these tools are as follows [1]:

- *ansysToFoam*: Converts an ANSYS input mesh file, exported from I-DEAS, to OpenFOAM® format.
- *cfx4ToFoam*: Converts a CFX 4 mesh to OpenFOAM® format.
- *datToFoam*: Reads in a datToFoam mesh file and outputs a points file. Used in conjunction with blockMesh.
- *fluent3DMeshToFoam*: Converts a Fluent mesh to OpenFOAM® format.
- *fluentMeshToFoam*: Converts a Fluent mesh to OpenFOAM® format including multiple region and region boundary handling.
- *foamMeshToFluent*: Writes out the OpenFOAM® mesh in Fluent mesh format.
- *foamToStarMesh*: Reads an OpenFOAM® mesh and writes a PROSTAR (v4) bnd/cel/vrt format.
- *foamToSurface*: Reads an OpenFOAM® mesh and writes the boundaries in a surface format.
- *gambitToFoam*: Converts a GAMBIT mesh to OpenFOAM® format.
- *gmshToFoam*: Reads.msh file as written by Gmsh.
- *ideasUnvToFoam*: I-Deas unv format mesh conversion.
- *kivaToFoam*: Converts a KIVA grid to OpenFOAM® format.
- *mshToFoam*: Converts.msh file generated by the Adventure system.
- *netgenNeutralToFoam*: Converts neutral file format as written by Netgen v4.4.
- *plot3dToFoam*: Plot3d mesh (ascii/formatted format) converter.
- *sammToFoam*: Converts a STAR-CD (v3) SAMM mesh to OpenFOAM® format.
- *star3ToFoam*: Converts a STAR-CD (v3) PROSTAR mesh into OpenFOAM® format.
- *star4ToFoam*: Converts a STAR-CD (v4) PROSTAR mesh into OpenFOAM® format.
- *tetgenToFoam*: Converts .ele and .node and .face files, written by tetgen.
- *writeMeshObj*: For mesh debugging: writes mesh as three separate OBJ files which can be viewed with e.g. javaview.

7.4 Closure

The chapter overviewed the realization of the FVM in a computer code by explaining several implementation attributes of the uFVM and OpenFOAM® CFD codes. The two codes were presented in terms of their data structure, memory management schemes, and case setup. The next chapter will detail the finite volume second discretization step as applied to the diffusion flux.

7.5 Exercises

Exercise 1

For the configuration shown in Fig. 7.11

(a) Write the owner and neighbor for each interior face and use it to write the owners and neighbors lists.
(b) Build the connectivity array for each element by looping over each of the interior faces using the owner-neighbor information; this is the coefficient connectivity used in uFVM.
(c) Build the coefficients ldu addressing connectivity used in OpenFOAM®.

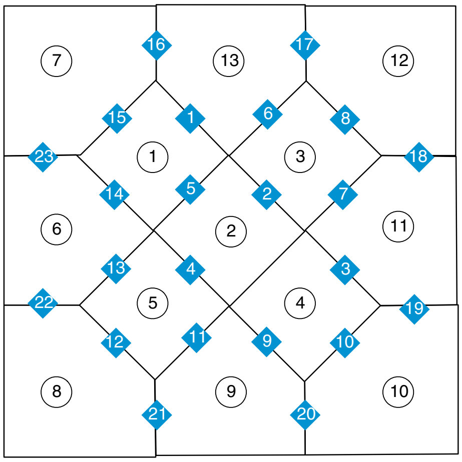


Fig. 7.11 The rectangular domain discretized with an unstructured grid used for Exercise 1

Exercise 2

Using the uFVM (MATLAB®) code, read the elbow mesh (available at the book website) and then do the following:

(a) Write a script to loop over all the elements of the domain and print out for each element its index, and the indices of its faces in the following format "element [i] → faces[k l m n…]"

(b) Declare and array phi as an element field (size = 1, Number of elements + Number of boundary faces), and initialize it using the following formula: $\phi(x, y, z) = 10xy + 5y^2$.

(c) Declare a new element array of type vector (size 3, Number of elements + Number of boundary faces) and use it to compute the gradient of phi, using the gauss theorem i.e.,

$$\nabla\phi = \frac{\sum\limits_{f} \phi_f \mathbf{S}_f}{V}$$

which can be written as follows for the mesh data used:

$$\nabla\phi = \frac{\sum\limits_{owner(f)} \overline{\phi_f}\mathbf{S}_f - \sum\limits_{neighbour(f)} \overline{\phi_f}\mathbf{S}_f + \sum\limits_{b=boundary(f)} \phi_b \mathbf{S}_b}{V}$$

Exercise 3

Use blockMesh to setup a uniform mesh similar to the one in Fig. 7.12 for $L = 1$. In OpenFOAM® and then in uFVM do the following:

(a) Write a program to read the mesh and loop over all boundary patches. Then for each patch print the centroid and normal vector of its faces.

(b) Modify the program to define a volumeScalarField T. Set the values for T at element centroids and at the boundaries to $10x^2y^2$, where x and y are the coordinates of the element centroids and for the case of the boundary the centroids of the boundary faces.

(c) Write a program to compute the gradient of T and compare its value with the analytical solution.

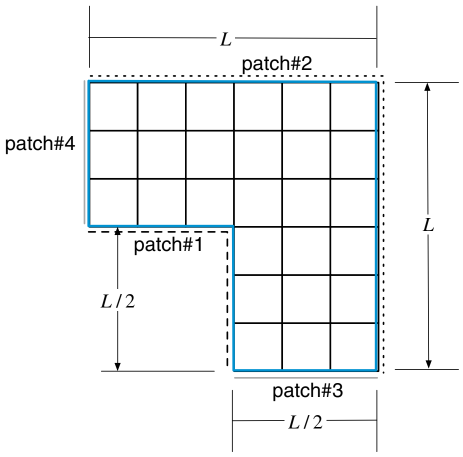


Fig. 7.12 A two dimensional domain generated with blockMesh

Exercise 4

(a) Find in OpenFOAM® using Doxygen [2], the class definition of the data types points, face, and cell.
(b) Loop over all interior faces (using forAll) and for each face write the owner, neighbor, and centroid.
(c) Loop over all elements and write for each element the value of the cubic root of its volume and its centroid.
(d) Loop over all boundary faces and write for each face the owner (parentCell) and centroid.
(e) Define a surfaceScalarField and set its value to be equal to the x component for interior faces and the y component for boundary faces.
(f) Find the member function of the surfaceScalarField (GeometricField<>) that returns the old time values.

References

1. OpenFOAM (2015) Version 2.3.x. http://www.openfoam.org
2. OpenFOAM Doxygen (2015) Version 2.3.x. http://www.openfoam.org/docs/cpp/