OpenFOAM Documentation Project
An Open Source initiative by:
Alberto Passalacqua
Jaswinder Pal Singh

Draft version for mesh classes documentation

primitiveMesh, polyMesh, fvMesh - The OpenFOAM mesh hierarchy

OpenFOAM Users

November 3, 2008

Open Source

Preface

In this document the major mesh classes of the OpenFOAM are explained and documented. The content of this document is based on knowledge acquired from the OpenFOAM discussion board. Most of the topics explained here has been already discussed on the forum.

The contents of this document will be open to a group of volunteers who will rectify any mistakes and refine the document successively over a time span. We have chosen LaTeX for this purpose due to its ease of use and popularity in the scientific community

Munich, August 2008

Contents

1 Primitive Mesh

It is important to understand what primitiveMesh is and how it is related to polyMesh and fvMesh, for the effective and advanced use of OpenFOAM. This is what Henry Weller said in one of his posts about these classes:

polyMesh, polyBoundaryMesh and polyPatch are generic, that is they do not presuppose any particular form of discretisation, they are the basic classes of the polyhedral mesh. fvMesh is derived from polyMesh and adds the particular data and functions related to the geometry needed for finite-volume discretisation. fvPatch is not derived from polyPatch but does hold a reference to one and also includes fuctions related to applying boundary conditions for finite-volume discretisation.

Before we explore the functional role of this class lets give a quick look of all its components. Broadly categorized this class has the following sections:

- private: (find a suitable sentence to describe these keywords. do not go into details)
  - permanent data - data permanently stored in the object corresponding to the class
  - private member functions - various topological and geometrical calculations and helper functions for mesh checking are declared in this section
  - static data members - these data members define static data which is used to control mesh checking
- protected:
  - - a null constructor "primitiveMesh( )" for the class is declared here
- public:
  - static data -
  - constructor from the components
  - a virtual destructor
  - Member functions - this section has basically all the functionality offered by this class

Next we give a look at each of the sections from functionality point of view. Grouping together the various member functions and the permanent data according to their functionality will help us to get a overall picture of what this class does. The information presented here is not new but just rearranged to grasp what this class does. Readers are encouraged to use the doxygen generated documentation of this class for details.

1.1 Permanent data

This section declares the permanent data which primitiveMesh holds. The nature of the data and its functional importance for the sub-classes such as polyMesh and fvMesh is what we want to understand at this point.

Consider that a set of points, also called as point cloud, is given. Now what kind of information is required if one wishes to join these points to obtain edges and connect these edges into a predetermined order to obtain faces. We can go one step further and group together the obtained faces to get cells. The cells can be of different shapes and for each shape we can define the configuration in terms of number of faces, edges and points and their connectivity.

It is obvious that a point or an edge or a face will belong to more that one cells. So how to obtain a mesh which is geometrically and topologically valid. The connectivity information is the key to the valid mesh. Once we have the valid mesh we might be interested in certain geometrical properties which are commonly required in the finite volume methods. Examples are cell volumes and the location of their centres. For face based operations we will need the face area vectors and their centres as well.

Now if we take a look at the entries in this section we will find that the declared data members facilitate namely this purpose.

1.1.1 Primitive size data

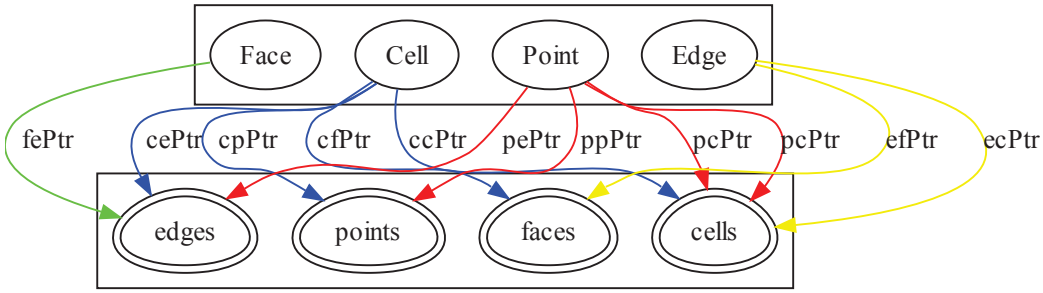
The section declares variables to hold the size related information. As mentioned earlier, points, edges, faces and cells are the basic primitives of a mesh. It is important to know the number of these elements in a mesh when we are declaring fields and lists to manipulate field data represented on a mesh. But why do we need the size of mesh primitives for the internal field. Basically the field data represented on a mesh can be further conceptually divided into field data on the boundary and field data internal to the domain. Such a division is necessary as we need to enforce various types of boundary conditions depending upon the physics. The figure shows the variables declared in this section:

1.1.2 Shapes

The description of shape configuration is predetermined. This information is held into a list of shell shapes. To identify the cell shapes constituting a mesh , we will need to check it against the cell shapes available in this list. To serve this purpose a pointer cellShapesPtr to the cellShapesList is declared. Next the access to edgeList is also necessary. Let me explain why. An edge constitutes a pair of points. In my opinion once we have obtained all the valid edges from a point cloud the resulting edgeList is further processed to obtain faces and cells. The ordering is applied to edgeList to form faces and cells subsequently. From this point of view an access to edgeList is required thus the pointer edgesPtr to edgeList.

1.1.3 Connectivity

All the variables in this section represent the connectivity information which is crucial to obtain a valid mesh from the given point cloud. The figure shows how the basic mesh primitives are interconnected. Each of these connectivity data represent the relation between a pair of mesh primitives. Consider the point-faces connectivity. Points are itself organized into a 1 dimensional labelList. As each point can be a part of number of faces it is essential to know how many faces share a point. Such kind of information can be stored for each point as a list for each point. The resulting data structure will be a list of lists. Each node of the original 1 dimensional labelList will store a labelList of faces to which the particular point represented at a node is connected. Observe that nearly all the connectivity information are represented by the list of lists. (Find out how the cellList differs from others).



1.1.4 Geometric Data

The variables declared in this section will hold the geometric data we need to work with the finite volume method. The data structures used to hold these data show the nature of information. Cell centres and face centers are vectorfields as a vector (x_i,y_i,z_i) is stored at each cell centre and face centre representing this information. The cell volumes are stored as a scalarField . A single value V_i is stored at each cell centre. Lastly the face area vectors represent the orientation of a face and their magnitude represent the area, holding such information in a vector field is obvious.

1.2 Private Member Function

In this part of the class header , the functions which are used for the internal processing of the permanent data are declared. An user of the class will have no direct access to these function. The user can call only the member functions available in the public interface. The member functions in the public interface do the necessary checking and then internally call the private member functions.

An user with working experience of OOP is aware of the fact that each class is supposed to have a copy constructor and an assignment operator. These constructors are required when one wants to assign an object of a class to another object belonging to same class or perform the copy operation on objects belonging to a class. C++ compiler creates these constructors implicitly if the user does not and these constructors have public access. OpenFOAM does not allow these opearations and thus these constructors are declared in the private section and thus access is removed. These constructors are just declared and never implemented thus an user trying to find their implementation in the corresponding C files will not find any code for these constructors.

Lets take a look at each of the subsections in this category and examine what they do.

1.2.1 Topological Calculations

This sections has the following member functions

- void calcCellShapes( ) const - Calculate cell shapes (next step - explain the function itself)
- void calcCellCells( ) const - Calculate cell-cell addressing (next step - explain the function itself)
- void calcPointCells( ) const - Calculate point-cell addressing (next step - explain the function itself)
- void calcCells( ) const - Calculate cell-face addressing (next step - explain the function itself)
- void calcCellEdges( ) const - Calculate edge list
- void calcPointPoints( ) const - calculate point point addressing
- void calcEdges(const bool doFaceEdges) const; - a lot to explain for this function, it is basically the workhorse of address calculation leading to the connectivity information
- void clearOutEdges( )
- static label getEdge(*, *, *, *) - given later the details of this function.

1.2.2 Geometrical Calculations

- void calcFaceCentresAndAreas( ) const - Calculate face centres and areas. It basically calls the function makeFaceCentresAndAreas(*, *, *). All the calculations are done in the called function, this functions checks for the debug switches and also creates the necessary data structures to hold information and then calls the makeFaceCentresAndAreas().

- void calcCellCentresAndVols( ) const - Calculate face centres and areas. Similarly it calls the function makeCellCentresAndVols(*, *, *,*). All the calculations are done in the called function, this function checks for the debug switches and also creates the necessary data structures to hold information and then calls the makeCellCentresAndVols().
- void calcEdgeVectors( ) const - explained separately
- void makeFaceCentresAndAreas(*, *, *) const - explained separately later
- void makeCellCentresAndVols(*, *, *, *) const - explained separately later

1.2.3 Helper functions for mesh checking

- checkDuplicateFaces -
- checkCommonOrder

1.3 Static Data Members

This group of static data members in the private area is declared to control the mesh checking parameters. They define the threshold values for mesh checking criterion used by the checkMesh utility. Theses values are defined as static variables as these are parameters unique for the class primitiveMesh and any object instance of this class should have the same value of these warning thresholds.

- Aspect ratio warning threshold
- Cell closedness warning threshold - set as the fraction of un-closed area to the closed area.
- Non orthogonality warning threshold in degrees
- Skewness warning threshold

1.4 protected

In this section only a null constructor for the class is declared. We will see later where it is used and come back here to provide more details about this constructor.

1.5 Static Data Members in the public section

The importance of static members has been discussed previously. As we already know that mesh primitives such as points, edges, faces and cells have an aggregative relationship this section provides a quantitative estimate of the relationship between a

pair of mesh primitives. The estimates assume a 3D cartesian mesh and all the values are based on this assumption. For example number of edges per face is 4 or take for instance number of points per cell, it is defined as 8. Going through the values and making couple of sketch one can see that the values are given for a 3 dimensional cartesian mesh.

1.6 Constructor

The constructor constructs the primitiveMesh from components. The constructor siganture reads

```
primitiveMesh
(
    const label nPoints,
    const label nInternalFaces,
    const label nFaces,
    const label nCells
)
```

This section will have the corresponding section from the C file as it shows how OpenFOAM uses the C++ initializer list to determine the initial values of the data members before the actual statements of the constructor body. The initial values can be set for every attribute in a list, separated from the constructor name by a colon. Familiarize yourself with the syntax of the direct initialization using initializer lists as OpenFOAM uses this overall and for the new comers it might be confused with the inheritance notation. The implementation of the primitiveMesh constructor is the best example to understand this construct. Once we are familiar with this simple format we will be able to understand the notation for the classes which are inherited from other classes and also class templates inherited from other class templates. One step at a time and we will be proficient with this notation. So here is the primitiveMesh constructor implementation:

```
primitiveMesh::primitiveMesh()
:
nInternalPoints_(0),    // note: points are considered ordered on empty mesh
nPoints_(0),
nInternal0Edges_(-1),
nInternal1Edges_(-1),
nInternalEdges_(-1),
nEdges_(-1),
nInternalFaces_(0),
nFaces_(0),
nCells_(0),
```

```
    cellShapesPtr_(NULL),
    edgesPtr_(NULL),

    ccPtr_(NULL),
    ecPtr_(NULL),
    pcPtr_(NULL),
    cfPtr_(NULL),
    efPtr_(NULL),
    pfPtr_(NULL),
    cePtr_(NULL),
    fePtr_(NULL),
    pePtr_(NULL),
    ppPtr_(NULL),
    cpPtr_(NULL),

    cellCentresPtr_(NULL),
    faceCentresPtr_(NULL),
    cellVolumesPtr_(NULL),
    faceAreasPtr_(NULL)
    {it does nothing in the constructor body}
```

Note that the order in which the attributes are listed in this list is same as the order in which the variables have been declared in the header file.

```
// Construct from components
// WARNING: ASSUMES CORRECT ORDERING OF DATA.
primitiveMesh::primitiveMesh
(
    const label nPoints,
    const label nInternalFaces,
    const label nFaces,
    const label nCells
)
:
    nInternalPoints_(-1),
    nPoints_(nPoints),
    nEdges_(-1),
    nInternalFaces_(nInternalFaces),
    nFaces_(nFaces),
    nCells_(nCells),

    cellShapesPtr_(NULL),
    edgesPtr_(NULL),
```

```
    ccPtr_(NULL),
    ecPtr_(NULL),
    pcPtr_(NULL),
    cfPtr_(NULL),
    efPtr_(NULL),
    pfPtr_(NULL),
    cePtr_(NULL),
    fePtr_(NULL),
    pePtr_(NULL),
    ppPtr_(NULL),
    cpPtr_(NULL),

    cellCentresPtr_(NULL),
    faceCentresPtr_(NULL),
    cellVolumesPtr_(NULL),
    faceAreasPtr_(NULL)
{
 //it does nothing in the constructor body
}
```

Note that in this case the constructor arguments which are passed on, are used to initialize the corresponding variable using bracket notation.

1.7 Destructor

Need to emphasize on the fact why the destructor has the keyword virtual.

The body of the destructor calls the member function clearOut( ) which clears all the geometry and addressing unnecessary for the CFD. In the body of this function the member functions clearGeom( ) (clear geometry related information) and clear-Addressing( ) (clear connectivity information) are called.

1.8 Member Functions - public

This is the major portion of the class header. Going through this in a systematic way will help us to identify the functionality which an user might be using often. Similar to the private section of member function, the functions in this section are grouped together according to their functionality and we will briefly discuss them in their respective groups.

1.8.1 Reset functions

Offers a twice overloaded reset function. The first one has the following signature:

```
code
```

It is used when the primitive array sizes are given. This is what happens when this function is called:

- clearOut() is called to clear all geometry and addressing information
- All the data members in the section *primitive size data* except nInternalPoints_ are reassigned
- Checks if the points are ordered and returns a bool(calls calPointOrder() - more on that separately later)
- Depending upon the bool value sets the value of nInternalPoints_
- prints a debug message if primitiveMesh debug switch is set to 1

The second version of this overloaded function is used when a reference to the cellList is given in addition to the array sizes. The signature reads:

```
code
```

This is what happens when this function is called:

- Calls the reset function with the first signature.
- sets the cell to faces connectivity represented by cellList pointer. The code sgement reads

```
cfPtr_ = new cellList(c, true);
```

1.8.2 Access

The purpose of access member functions is to provide the class user with the access to the information contained in the private and protected data members. Also note that all these functions are inline functions and have the keyword const at the end. These are explicitly defined as inline functions. What is the benefit of inline functions ?. The keyword const implies that these functions do not change the information.

1.8.3 Primitive Mesh Data

In this section , all the functions are declared as the pure virtual function. Take for example:

```
virtual const pointField& points() const = 0
```

With this syntax, one declares a pure virtual function. The functions in this section are implemented in the inherited class. We will got through their implementation in the class polyMesh.

1.9 Derived Mesh Data

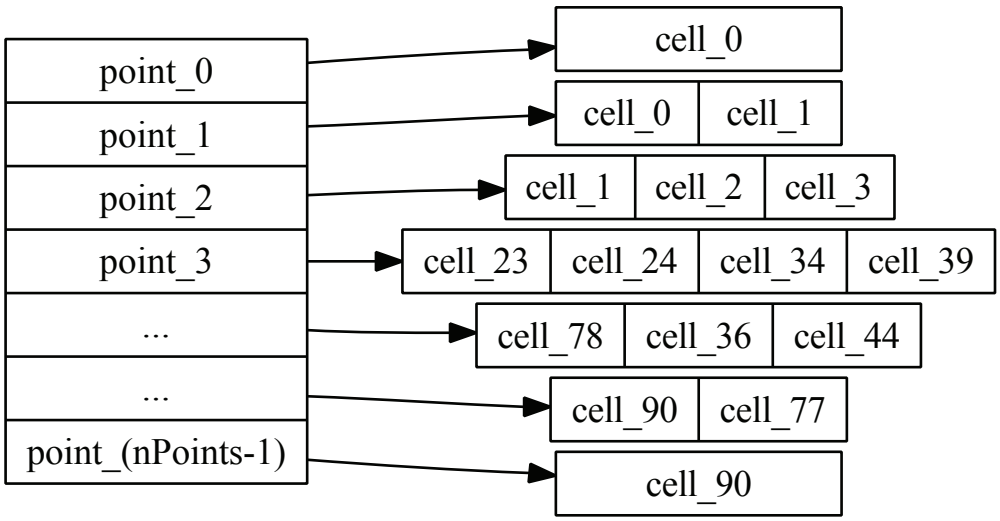
1.9.1 Return Mesh Connectivity

Efficient organization of the information related to the connectivity of various mesh primitives is very important. We are aware of the private member functions which calculate the connectivity addressing between the basic mesh primitives. As an user have no direct access to these functions, a public interface to these functions is provided in this section. Grouping together the functions according to the mesh primitives will give us a better overview.

1.9.1.1 point connectivity

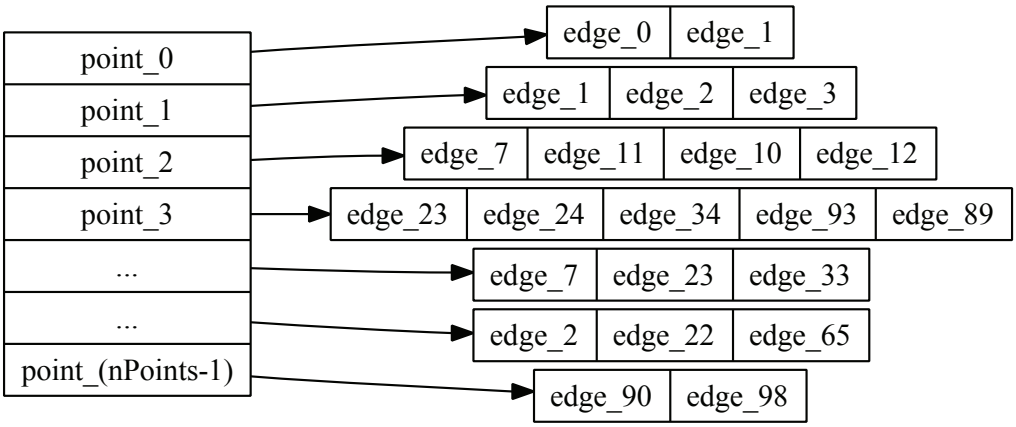
These are the function providing the point connectivity to other mesh primitives:

- pointCells - gives the information how many cells share a point. This function just checks whether the data structure to hold the information already exists and then calls the calcPointCells( ) which basically does the actual work.



- pointEdges - gives the information how many edges share a point. The information returned in the form of list of lists has at each node of the list corresponding to a point , a list of edges sharing this point. Similar to the 1st function it does

the necessary checks and calls the calcPointEdges() from the private member



function.

- pointPoints - gives the information how many points are connected to this point through edges originating / ending at this point. The call is made to the calcPointPoints( ) after the routine checks. The calaPointPoints( ) loops through the point to edged connectivity labelListList. We know that each node of this point list contains a list of edges. All this routine does is that it loops through the edge list stored at each node and checks whether the point corresponding to the node under examination is the start point (edge.start()) or end point of the edge (edge.end()). If edge.start() == pointI (current value of point list iterator) then get the edge.end() and store it into the point-points connectivity list else if edge.end() == pointI then get the edge.start() and store it into the connectivity list. (we need here a clearer explanation or else a clever picture which explains this concept of list of lists browsing)
- pointFaces - It is quite obvious by now that this gives us list of faces sharing a point. The information is, once again, stored in a list of lists data structure. To obtain this information an inverse operation is applied to the faceList (a 1-D List which stores the face information at its nodes). The function call, which does the job, looks like this:

```
invertManyToMany(nPoints(), faces(), *pfPtr_)
```

(We will try to find out how this works while extending this document. Please email us if you have know how that works)

1.9.1.2 cell connectivity

These are the function providing the cell connectivity to other mesh primitives:

- cellCells -
- cellEdges
- cellPoints

1.9.1.3 edge connectivity

These are the function providing the edge connectivity to other mesh primitives:

- edgeCells -
- edgeFaces -

1.9.1.4 face connectivity

These are the function providing the face connectivity to other mesh primitives:

- faceEdge -

1.9.2 Geometric Data

The functions under this section are responsible for processing geometrical information. Similar to the functions under topological information these functions are public interface to the corresponding functionality available in the private section. The inherited classes primitiveMesh and fvMesh will use these function signatures to get the information. The functions read :

- cellCentres( )
- faceCentres( )
- cellVolumes( )
- faceAreas( )

1.9.3 Mesh Motion

The sole function in this section is used for dynamic mesh functionality. Given the pointFields at time (t) and at time ($t-1$) it will return the volume swept by the faces in motion as a scalarField. The implementation is quite simple. It does some size checks on the pointFields supplied as arguments. On success it then iterates upon the faceList and for each face in turn calls the function face.sweptVol(oldPoints, newPoints). The class face has the implementation of this member function. (We will provide the working details of it under the documentation of mesh shapes). Just for the record the signature of the sole function in this reads:

```
tmp<scalarField> movePoints
(
    const pointField& p,
    const pointField& oldP
);
```

1.9.4 Mesh Checks

The class primitiveMesh has all the basic mesh checking functionality implemented which has been divided into two categories:

- Topological Checks
- Geometrical Checks

What is being checked under these categories has been briefly commented upon in the sub sections. In addition to these individual checks the primitiveMesh class has certain member functions which group together the checks under a particular category. checkTopology( ) - as its name suggests, calls the functions listed under the sub-section topological checks and checkGeometry( ) - naturally invokes the geometrical checks. The function chekMesh( ) consolidates all this checking business into a single function call, which an user can off-course call without knowing the details.

The function checkMotion - still needs some understanding and formulation for explanation.

1.9.4.1 Topological Check

In this section functionality, for various topological checks, is provided. From topological point of view it is important that certain mesh primitives such as cells and faces comply to specific requirements. Consider the cells first, for this particular mesh primitive it is essential that it is topologically closed. (We need some input here as I did not understand what exactly the corresponding code is doing here. Has to do something with the single edges. Anybody who knows what exactly is happening here, please communicate).

The faces have to comply to more checks for them to be topologically valid. This requires that the vertices of a face are valid and unique i.e. the list of labels defining a face does not have two labels with the same label_ID and no label_ID exceeds the number of points returned by nPoints(). The face-face connectivity check performs various checks on the common points.(Detail of this member function - next step). Face ordering checks whether internal faces are ordered in the upper triangular order. (it would be nice to have an explanation what does upper triangular order means).

The functions in this section thus are:

- checkCellZipUp( ) - for cells
- checkFaceVertices( ) - applies to faces
- checkFaceFaces( ) - applies to faces
- checkUpperTriangular( ) - applies to faces

1.9.4.2 Geometrical Checks

There are numerous geometrical checks for the basic mesh primitives in this class. The function names are self explanatory of their functionality (the details of their implementation is our next step). They have been organized below in a tabular form for an overview.

cells	**faces**	**edges**	**points**
cell closedness negative cell volumes cell determinant	negative face areas non-orthogonality face pyramid volume face skewness face angles face flatness	edge alignment edge length	unused points point-point nearness

1.9.5 Useful derived info

The functionality provided by the member functions in this section is commonly required. The functions are:

- bool pointInCellBB(const point& p, label cellI) const
- pointInCell(const point& p, label cellI) const
- findNearestCell(const point& p) const
- findCell (const point& p) const

Lets take a look at how each one can be used. Consider that we create an object "mesh" of class fvMesh:

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

and an object "location" of class point:

```
point location(0.5, 0.25, -0.25);
```

1.9.6 Storage Management

The functions in this section take care of the memory management. Their main purpose is to free up the memory at the end of the program life cycle. The functions read:

- printAllocated() - This function provides a report on all the allocated labelListLists which contain the connectivity related information.
- clearAddressing() - as the name suggests this function calls the delete() for all the connectivtiy related alloacted structures
- clearGeometry() - deletes all the geometry related fields, namely the ones which contain the cell volumes and centres and face area vectors and face centres.
- clearOut() - groups together the above listed clearAddressing() and clearGeometry in a single function call and is call form within the body of the destructor.