CHALMERS

Implementing Anisotropic Adaptive Mesh Refinement in OpenFOAM

Master's Thesis in Computer Science, Algorithms, Languages and Logic

JONAS KARLSSON

Department of Computer Science
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012
Master's Thesis 2012

Implementing Anisotropic Mesh Refinement in OpenFOAM

Jonas K. Karlsson

Examiner: Peter Damaschke

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden December 2012

Abstract

The field of computational fluid dynamics (CFD) is growing steadily, with a rate proportional to the computational tools available. With today's computing power available in CPU's and clusters we are able to solve cases that were considered impossible just a few years ago. Although as we all know, even with great computational power and brute-force approaches, some problems will remain unsolvable. With this realization, adaptive mesh refinement (AMR) was introduced as a way of adapting the mesh as to reduce the computational error. AMR is relevant for CFD since it can greatly reduce the computational effort needed to solve a lot of cases. This can in turn make previously intractable simulations solvable. In this thesis we will look at how implementing a specific type of AMR could be done - namely anisotropic AMR for the OpenFOAM code-base.

We will briefly look at some papers to get an idea of what constraints the mesh should satisfy and to see what kind of data-structure for the refinement history is needed. We will also look at similar functionality currently available in the OpenFOAM code. With this knowledge we define a design criteria we will use for implementing anisotropic AMR in OpenFOAM.

This implementation will result in a working example of anisotropic AMR. We have also defined a tree used for storing the refinement changes that can be used for other types of AMR in the OpenFOAM codebase. We also show some graphical results of a case refined using the anisotropic AMR defined in this thesis and compare these to an isotropic refinement.

Acknowledgements

I would like to thank my supervisors Claes Eskilsson and Andreas Feymark for their great support and help during my thesis work. I would also like to thank Peter Damaschke for his supervision and also Matz Johansson for his input during the opposition. Lastly I would like to thank all my friendly colleagues in the Hydrodynamics Group at the Department of Shipping and Marine Technology, Chalmers.

Jonas Karlsson, Göteborg 14/10/12

Contents

Chapter 1

Introduction

In the field of fluid mechanics the introduction of Computational Fluid Dynamic (CFD) has caused a paradigm shift in how a great deal of the fluid research is executed. CFD can be defined as [1]:

> "the science of computing numerical solutions to Partial Differential or Integral Equations that are models for fluid flow phenomena"

But the importance of CFD stretches further than just providing insights into the physics of fluids, throughout the years it has evolved to become an important tool for a plethora of engineering disciplines. Today there is plenty of CFD software available, some of the most well-known being ANSYS-CFX, ANSYS-Fluent, CFD++, NUMECA and Star-CCM+, just to name a few. It is common, and almost mandatory, for CFD applications to execute in a parallel environment typically on computer clusters. The effect of this is that the licensing cost for the used applications can become quite significant. Therefore there is a need and interest for alternative, and in particular open source, solutions. Of course, price is just one factor that makes it interesting, the possibility to modify the code is equally important. Arguably one of the most successful open source CFD tools is, in terms of user adoption, OpenFOAM® [1] [2]. The details of OpenFOAM are described in chapter 2, for now we only mention that OpenFOAM is based on a numerical technique called finite volume method (FVM).

FVM is a mesh (or grid) based method where the computational domain is subdivided into several cells, together creating a mesh. The truncation error from a numerical solution can be seen as $e_n = O(h^p)$, where e_n is the truncation error, h represents the cell size and p denotes the order of accuracy of the numerical scheme [3]. From this equation it is obvious that to reduce the error you need to either decrease the cell size or increase the order of accuracy. Decreasing the error is crucial if you want to achieve an accurate computation of the flows involved in the simulation.

[1] Te work described in this thesis is not approved nor endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.

When representing a mesh in a computer environment there are two basic classifications that are based upon the connectivity of the mesh, namely structured or unstructured [4]:

- Structured (curvilinear). A structured mesh is a mesh that follows some regularity and that can store its points connectivity as an n-dimensional matrix. For example in a two-dimensional mesh an array $x(i,j)$ can store every points x-coordinate. Then i and j can be selected to represent the position of points in one curvilinear direction. With this approach any curvilinear coordinate system will also conform into a structured mesh. Naturally a hexahedral uniform mesh is easiest to imagine stored in a structured grid, but other less trivial meshes can also be stored in the same structure. A common example used for understanding the implications of a curvilinear mesh is to imagine a sponge with a uniform hexahedral mesh drawn upon it. Then you could take this sponge and wrap it around objects, for instance around a cylinder, whereby the mesh connectivity still will be the same but the actual cells will be curved. By forcing this regularity the well-representable meshes are reduced to quadrilaterals (in 2D) and hexahedra (in 3D).
- Unstructured. An unstructured mesh is irregular in the sense that it cannot be readily stored in an array or a matrix. Compared to a structured mesh an unstructured mesh can represent any type of cell since it is not restricted to keep the connectivity representable in a matrix. The downside is of course that the connectivity needs to be stored explicitly, any vertex can have an infinite (although not in practice) number of connections.

1.1 Adaptive mesh refinement

When performing a computation for a general CFD case it is impossible to know a priori how to design an optimal mesh, i.e. a mesh with minimal number of cells still satisfying the defined tolerance of the computational error. For transient problems, where the flux is unsteady and the points of interest can reposition during the simulation, this becomes most problematic since a static uniform mesh would need to be very fine to satisfy the error tolerance throughout the whole simulation. To solve this dilemma a scheme where the mesh self-adapts its structure upon some criteria can be used. What this usually means is that the mesh initially is divided uniformly and during runtime the mesh is automatically altered. The criteria is commonly an error-estimation which will mark cells with error greater than some threshold to be refined and likewise mark cells with an error lesser than the threshold to be unrefined (if once refined). The effect of this scheme is a more effective use of cells and thereby lower computational cost. There are three basic schemes available for adapting the numerical solution:

- h-adaptivity. In h-adaptivity the mesh connectivity is changed. The mesh is refined by adding points, thus reducing the size of the cell. The mesh is coarsened by deleting points. In this method the mesh connectivity is changed.
- p-adaptivity. Adaptivity is obtained by increasing or decreasing the order of accuracy of the underlying numerical scheme. In this method the mesh connectivity remains unchanged.

- *r*-adaptivity. Both the mesh connectivity and order of accuracy of the scheme are kept constant but the points are repositioned to minimize the computational error.

The most common scheme is *h*-adaptivity, also known as adaptive mesh refinement (AMR). In this study only AMR is considered. With AMR the overall error can be reduced using less cells compared to a uniform refinement, which in turn leads to greater efficiency in terms of memory and computational effort. Because of this, some problems that are intractable to solve with a uniform grid can become solvable when using AMR [5].

When performing AMR there is an important distinction between isotropic and anisotropic mesh refinement. In isotropic mesh refinement, from the Greek *iso* (equal) and *tropos* (direction), the mesh is refined in all directions. This means that all cell edges (or sides in 3D) are split, i.e. one quadrilateral is split into four quadrilaterals and a triangle is split into four triangles. Isotropic mesh refinement guarantees that the mesh quality is not degraded and that the underlying matrices stay well conditioned. Isotropic refinement is what the distributed version of OpenFOAM supports.

Anisotropic mesh refinement is, as the name might suggest, refinement in only one direction, so for instance a hexahedron gets split into two. In many CFD simulation there is often a pronounced directional behaviour in the flow field investigated. Two examples of where these behaviours will arise is on boundary layers (on the boundary of the computational domain) and with shock waves, where the flow varies dramatically only in the normal direction of the shock or layer. For these scenarios it is more computationally efficient to use stretched cells refined in only the direction of the shock so to reduce the error with less cells. These types of cells are possible to obtain by using a coarse initial mesh and then applying anisotropic refinement. Typically, the number of cells given by an anisotropic refinement is only a fraction of the number of cells given by an isotropic refinement, see Figure 1.1.

When refining one cell but not its neighbour cell the face between the two cells will get split, however this results in that one of the cells will have seven faces (in the case of anisotropic refinement of a hexahedron). A point is called regular if it constitutes a vertex (corner) for each of the neighboring cells; otherwise it is irregular [7]. In the context of two-dimensional meshes, the maximum number of irregular nodes on an element side is referred to as the index of irregularity. Meshes with an index of irregularity equal to one are called 1-irregular meshes. In this case the point connecting the neighbour and the two refined cells is called a “hanging node”, see figure 1.2. This does also extend to three dimensions where in the case of an anisotropic refinement we will have 2-irregular meshes, that is meshes with a maximum of two irregular nodes on each side. From this we define a *consistent mesh* as a mesh where every side of a cell comprise at most two faces. This is also commonly referred to as the “two-to-one” rule. There are several practical and theoretical reasons to restrict the mesh to 1-irregularity, especially in the context of *h-p* refinement, for more details see [8].

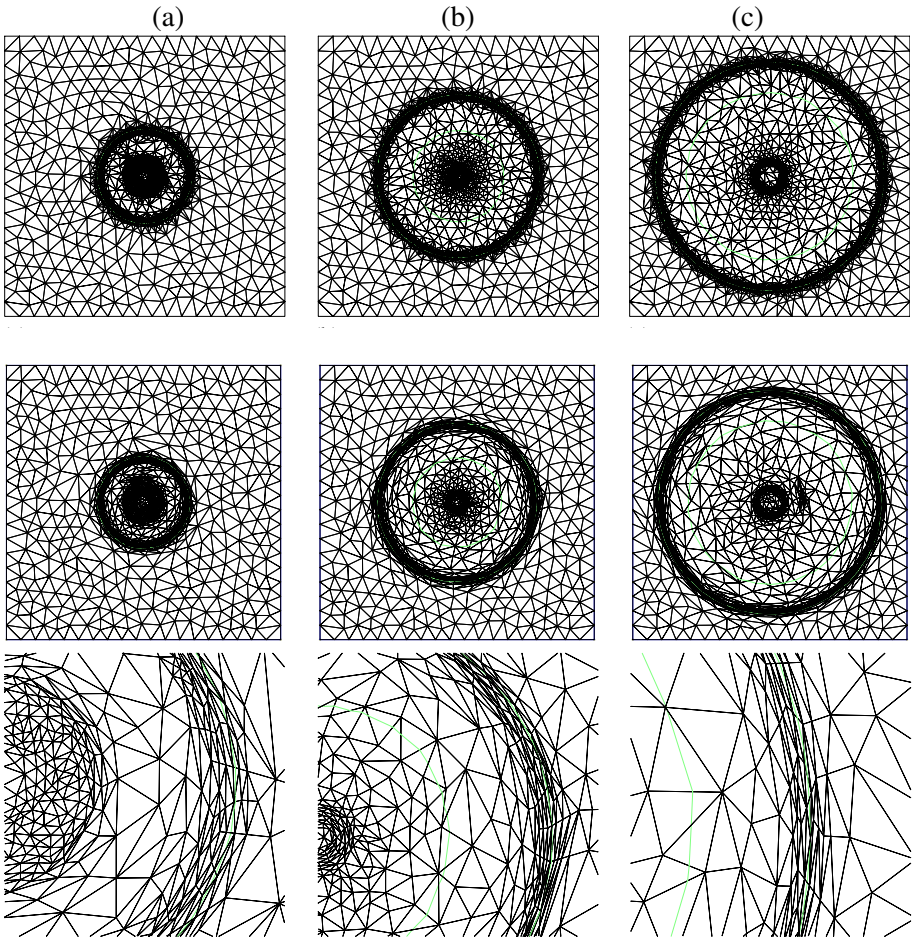


Figure 1.1: Computations of a radial dam-break using isotropic mesh refinement (top panel) and anisotropic mesh refinement (middle panel, and a zoom in the lower panel). The computations have the same accuracy but have very different degrees of freedom: (a) $t = 0.5$ the isotropic has 5386 elements while the anisotropic has 2078 elements, (b) $t = 1.0$ the isotropic has 7198 elements while the anisotropic has 1920 and (c) $t = 1.5$ the isotropic has 9510 elements while the anisotropic has 1932 elements. The variable t is the elapsed time. From [6].

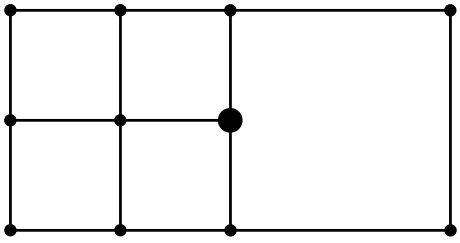


Figure 1.2: An example of a hanging node in 2D. The dots represent nodes and the hanging node is represented by a larger dot. This mesh is 1-irregular.

1.2 Parallelization

In CFD, parallelization has emerged as one of the corner stones to its success. For computer clusters the most common way to distribute computations among several processes is through a Message-passing Interface (MPI) [9, Ch. 8]. This can be used to do domain decomposition, that is decompose the domain into n-sized parts and then send these with MPI communication to n different processes. The term 'process' is quite arbitrary and can be either a different node on a network, or a different core on the same CPU. A more general term for this is load-balancing which is what a method used for distributing the workload between several different processes is called. Generally the benefit is that several processes working on small chunks of a problem will make the computation complete faster than just one processor handling the whole computation, thus also enabling simulations of much larger scale. Dividing the mesh in too small chunks might make the communication cost higher than what is gained by dividing the computation so there are drawbacks too.

OpenFOAM provides support for parallelization through MPI. By default OpenFOAM ships with the OpenMPI library. OpenFOAM supports load-balancing and decomposition through some third-party applications like metis[10] (and the parallel version parmetis) and scotch[11] (and the parallel version ptscotch).

AMR will typically cause initially well-balanced distributions to become unbalanced as new cells often are clustered on one or a few processors. For instance if one processors decomposition contains a vortex it will likely produce many more new cells. In the context of AMR, load-balancing should be performed automatically during run-time in order to redistribute the mesh after some steps in time or a certain amount of refinement has been done. This is known as dynamic load-balancing. The vanilla version of OpenFOAM does not currently support dynamic load-balancing, but the present work is based on an in-house version of OpenFOAM-2.1.x at the Department of Shipping and Marine Technology [12] that supports dynamic load-balancing, see Figure 1.3

1.3 Refinement tree

When using AMR a data-structure to hold the refinement changes is needed to be able to do unrefinement. It is also necessary to have a distributable refinement-tree in a parallel environment to be able to distribute refined meshes. One commonly used type of refinement is called octree-refinement. It is based upon the notion of octrees where the nodes of the tree represent cells in the mesh. This method works by creating a root node (or bounding box) for the whole domain and then split this node into eight new sub-nodes, hence creating an octree. Every node represents a cell and also inherits a geometric placement based upon its position in the tree. For instance, if the whole domain is split into eight cells then every cell contains one eight of the domain. If any of these cells get split then their children will contain one sixty-fourth of the domain.

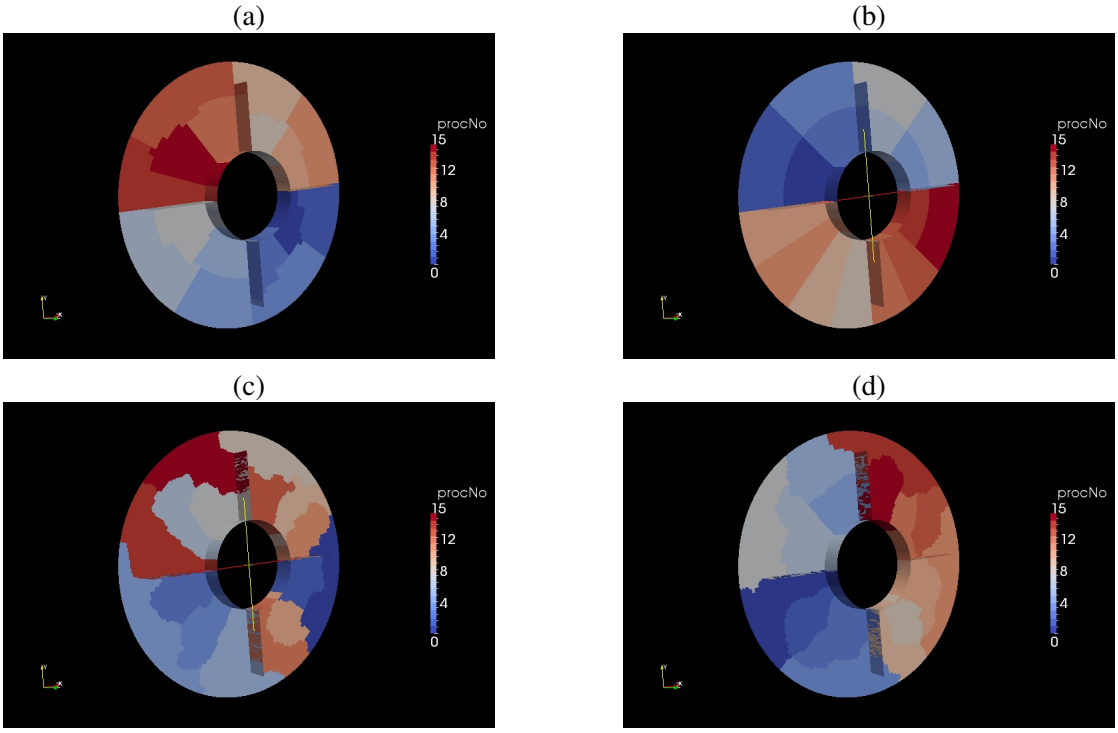


Figure 1.3: Dynamic load-balancing of a rotating mixer: (a) initial partition using metis, (b) initial partition using scotch, (c) final partition using metis, and (d) final partition using scotch.

Mitchell proposes a data-structure called REFTREE (or *refinement-tree*) used for AMR [13]. The *refinement-tree* $T(G) = \{V, \{C(v_i)\}\}$ contains a set of nodes $V = \{v_i\}_{i=0}^{M}$ and each $v_j \in V$ containing a set of children $C(v_i) \subset V$ where G is the grid. Every node is contained in exactly one set of children $C(v_i)$ except for the root node which is not contained in any. If $C(v_i) = \emptyset$ then v_i is called a *leaf*. If we call the root-node v_{root} then $C(v_{root}) = G_0$ where G_0 is the initial mesh (without any refinement). Figure 1.4 shows an example of an oct-tree and refinement-tree.

1.4 Objective

OpenFOAM has support for isotropic AMR, limited to octasectal refinement of hexahedra. Even though there are examples of studies using anisotropic AMR in OpenFOAM [5], to the author's knowledge this functionality is not presently distributed. This work aims to provide OpenFOAM with support for anisotropic mesh refinement, in order to increase the computational efficiency when using AMR. The scheme should work in a parallel environment. In order to achieve this, functions need to be created that:

- Select cells to refine and the direction in which to split them.
- Split selected cells anisotropically.

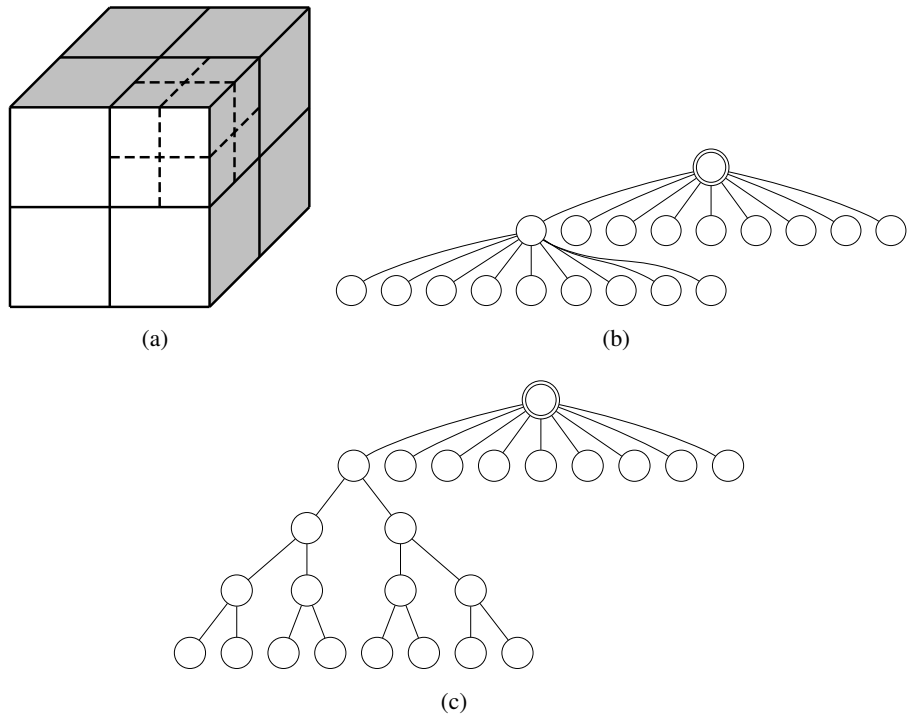


Figure 1.4: A three-dimensional mesh (a) and its oct-tree (b) and its anisotropic refinement-tree (c) representation. The dotted lines means that the cell is refined. Note that the refinement-tree in this example does not represent the same refinement as the oct-tree but instead an anisotropic refinement.

- Store the refinement changes in a refinement-tree.
- Distribute the refinement-tree.
- Automatically refine and redistribute the mesh and tree.

The work is restricted to only handle hexahedra. Some challenges of implementing AMR is that a geometric orientation needs to be created. This orientation is needed to be able to know how faces, edges and points should be modified by a split. Since the mesh must be consistent we also need to define exactly what this consistency means for a cells ability to be refined and also what preconditions have to be met for cell to be refineable. Since the mesh is distributable this geometric orientation for the cells will also need to be distributable. Overall time-complexity is crucial in every part of the system, as simulations might run for extended periods of time, an inefficient time-complexity is not acceptable.

1.4.1 Design Criteria

The implementation is limited to function only for hexahedra and it should work in a parallel environment. The implementation should be able to select candidate cells for refinement

by some arbitrary error estimation and, if possible by the refinement constraints, refine these cells. A refinement-tree needs to be implemented to be used together with the application. The refinement-tree needs to satisfy the following criteria:

- Handle any type of polyhedral cell.
- Function in parallel and handle distribution between processes.
- Efficient in terms of memory and computation cost.

By making the tree handle any type of cell it should thus be implementable in the current OpenFOAM codebase for octasectal hexahedra refinement.

Chapter 2

OpenFOAM

This chapter will give a description about the design of OpenFOAM regarding how it handles AMR. Figure 2.1 shows the overview of OpenFOAM. OpenFOAM (Open Source Field Operation and Manipulation) is a library of tools, written in C++, that are used primarily for creating applications. The tools fall into two different categories; solvers and utilities. Solvers are used to solve continuum mechanics and the utilities are used for pre- and postprocessing. This modular structure makes it possible for users, given they posses the ability, to combine and program their own solvers and applications. The preprocessing part involves tasks for setting up the case like; mesh creation, mesh conversion (from third-parties) and defining numerical schemes. Postprocessing usually involves visualization of the case, for this the freely licensed open-source application paraFoam is provided but other proprietary tools like EnSight can be used by converting the case data to suitable formats.

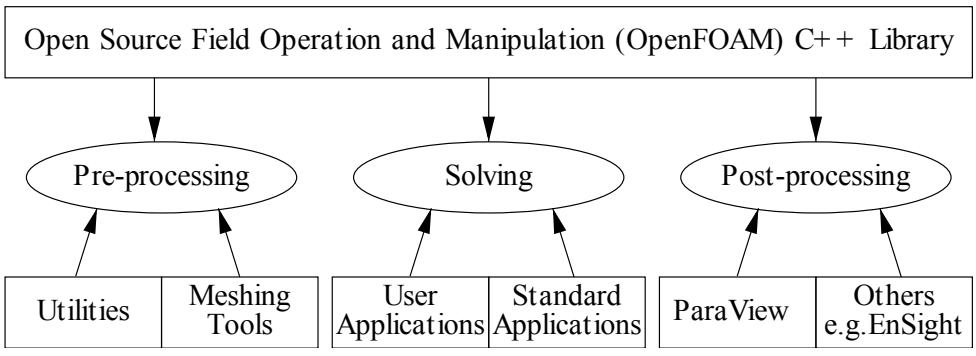


Figure 2.1: Overview of OpenFOAM [14].

2.1 Mesh description

In OpenFOAM a mesh is described by four basic building blocks:

- Points. A point defines a location in 3D-space by a vector holding its coordinates. Two points are never allowed to have identical coordinates. A point must also belong to at least one face.
- Faces. A face is an ordered list of points where every point is represented by its label.
- Cells. A cell is a list of faces represented by their labels.
- Edges. From the field of geometry, represents a line segment connecting two points. The connecting points are references by their labels.

Every point, face, cell and edge is assigned a unique label representing only that particular object. The most basic mesh is implemented in the class `primitiveMesh` which represents the mesh described above. This class also provide som other useful functionality to the mesh which is described a bit more in Section 2.2.1.

2.2 Adaptive mesh refinement

The implementation of AMR in OpenFOAM depend upon three key classes; `dynamicRefineFvMesh`, `hexRef8` and `refinementHistory`. The class `dynamicRefineFvMesh` holds the mesh and gets called to do mesh refinement (*refinement-engine*), `hexRef8` is used for cutting cells (*mesh-cutter*) and `refinementHistory` contains the history of the refinement (*refinement-tree*).

2.2.1 Refinement-engine

Figure 2.2(a) shows the inheritance diagram for `dynamicRefineFvMesh` which is a subclass (through several subclasses) of `primitiveMesh`. Through this inheritance `dynamicRefineFvMesh` has been equipped with functions to handle polyhedral cells and to do finite volume discretization. To fully understand the implications of this inheritance a breakdown of the classes used is needed to understand their purpose.

The class `primitiveMesh` is as the name implies the most primitive form of a mesh. This also means it actually contains the permanent mesh data and that it provides the functions to create the connectivity structures of the mesh as well as some geometric data. The geometric data that `primitiveMesh` provide is functions to calculate cell centres, cell volumes, face centers and face areas. The connectivity structures contains for instance; which cells and edges a point belong to or which cells are connected to a certain cell. To get the points and cells into its structure `primitiveMesh` provides the virtual (overriding) functions `points()`, `faces()`,

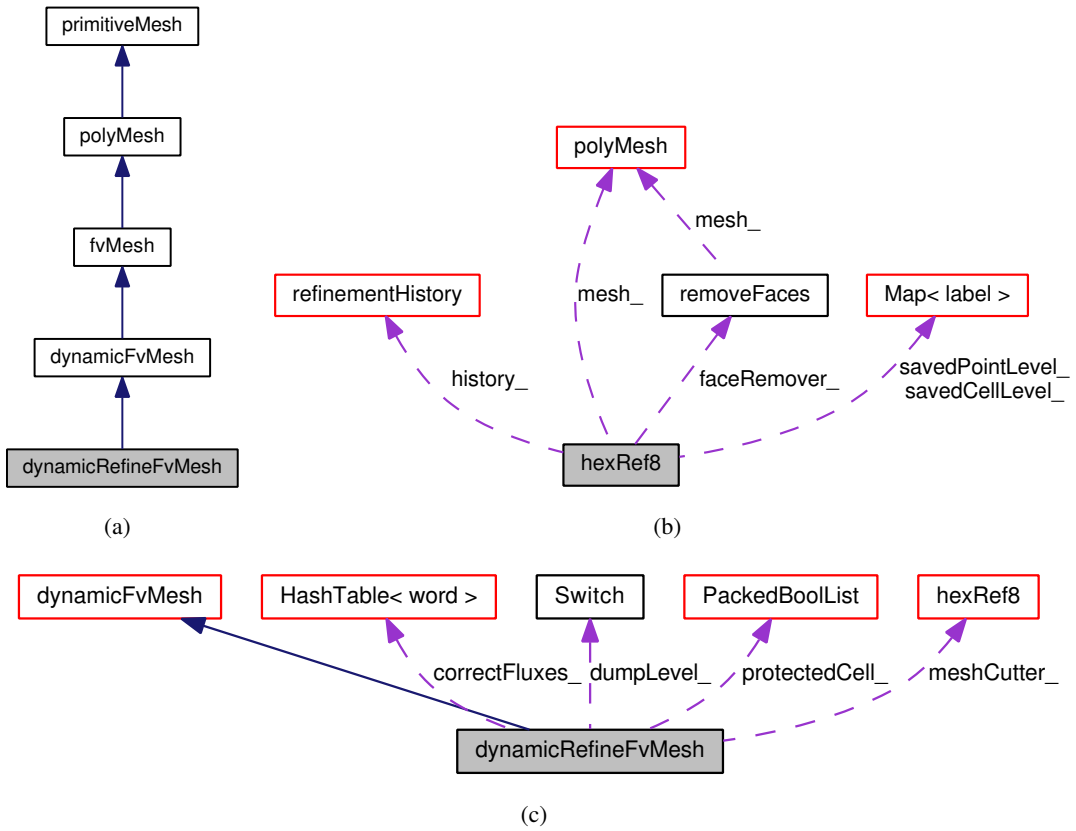


Figure 2.2: The inheritance diagram for `dynamicRefineFvMesh` is shown in (a) and the collaboration diagram is shown in (c). The collaboration diagram for `hexRef8` is shown in (b).

`faceOwner()` and `faceNeighbour()`. Since these functions are virtual the subclasses will have to implement them. This brings us to the next class in the inheritance hierarchy which is `polyMesh`. The class `polyMesh` can be seen as a mesh of generic polyhedral cells. It provides the override functions described before; `points()`, `faces()`, `faceOwner()` and `faceNeighbour()`. When constructing the `polyMesh` object the structures for points, faces and cells can be read from files in the current case directory. So at this level a proper mesh is actually contained. The next subclass is `fvMesh`. The class `fvMesh` adds the functionality to do finite volume discretization. It adds among other things fields for the face area motion fluxes, face area vectors and face area magnitudes. The next class is `dynamicFvMesh` which is just an abstract base class for geometry- and/or topology changing meshes. It provides just one function besides the constructor and destructor which is a virtual function called `update()`. Now we have arrived at the class doing the mesh refinement, namely `dynamicRefineFvMesh`.

This class provides the function `update()` which for this class executes an adaptive refinement on the mesh. The update function uses some other internal functions like `selectRefineCandidates()` for selecting the candidates for refinement and `refine()` for doing the refinement of some selected cells. To do the actual cutting of cells `dynamicRefine-`

FvMesh uses another class called hexRef8 stored in the variable meshCutter_.

2.2.2 Mesh-cutter

The class hexRef8 is called a mesh-cutter and is used for splitting hexahedral cells into eight subcells. The cutting of cells is done by adding mesh modifications to an object of the class polyTopoChange. The class polyTopoChange has functionality for adding, removing and modifying points, faces and cells. The class hexRef8 contains a function setRefinement() that takes as input a list of cells and a polyTopoChange and then insert the refinement changes of these cells into polyTopoChange. From figure 2.2(b) we can see that hexRef8 contains a variable history_. This variable is of type refinementHistory and is the refinement-tree for the mesh. Since hexRef8 is a mesh cutting engine for splitting hexahedral cells into eight sub-cells OpenFOAM is by default limited to only this type of refinement. Figure 2.2(c) shows the collaboration diagram for dynamicRefineFvMesh. From the diagram we can see that to be able to implement AMR for anisotropic refinement we need to replace hexRef8 with a mesh-cutter capable of doing anisotropic refinement.

2.2.3 Refinement-tree

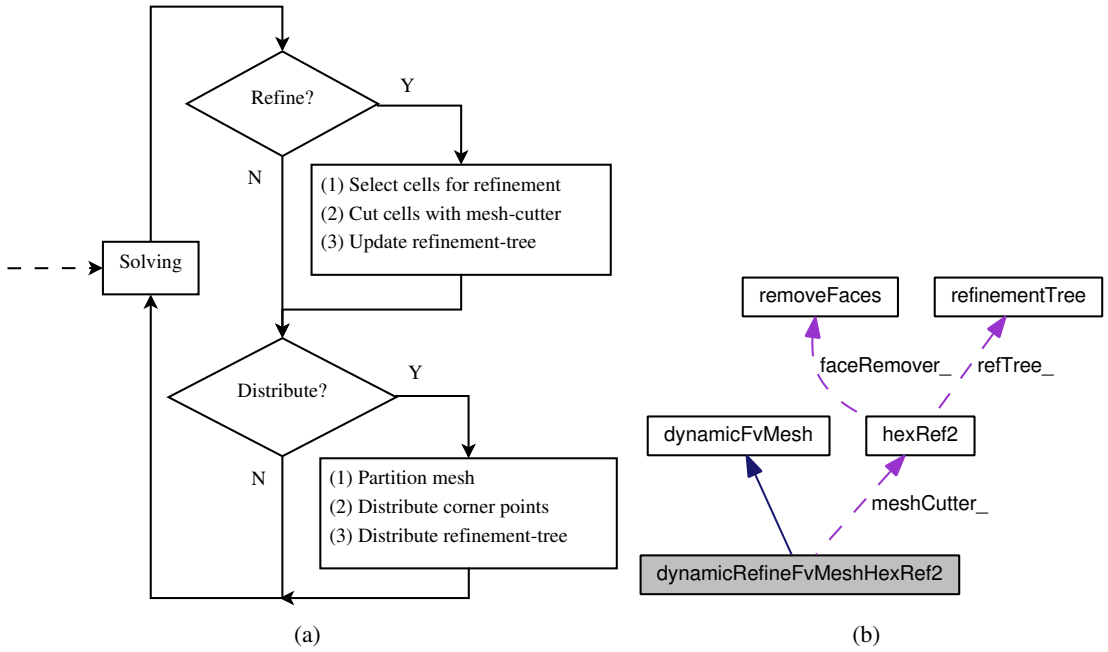
The class refinementHistory is the class containing the history of the refinement done so far. This history shows how the current cells in the mesh have come to be, i.e. it shows for every cell which cell it was refined from. Likewise, if a cell has been refined, the history also show which cells it was refined into. This class provides functions to update the history and to redistribute the history to other processes. Note that this class only stores the labels of the cells and their refinement history, no points nor other values are stored. This refinement history is just like hexRef8 limited to only octasectal refinement. To do anisotropic refinement refinementHistory needs to be replaced by a tree capable of handling at least anisotropic refinement, but preferably capable of handling any type of refinement.

Chapter 3

Implementation of AMR

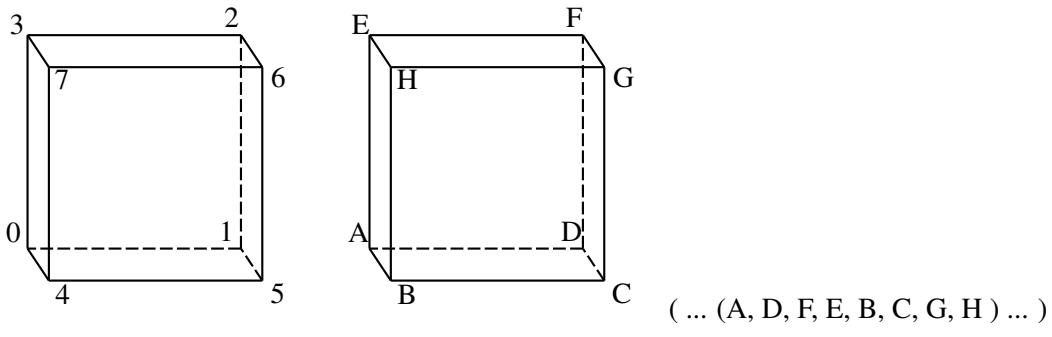
As we saw in the previous chapter we need to create a mesh-cutter like the class `hexRef8` but capable of doing anisotropic refinement, this new class is called `hexRef2`. The class `refinementHistory` is replaced with a refinement-tree capable of representing any type of AMR. Since this tree is independent of the type of refinement executed it is compatible with both `hexRef2` and `hexRef8`. Ideally we would have been able to use the refinement-engine `dynamicRefineFvMesh` for both anisotropic and isotropic refinement but since its implementation was designed with regards to isotropic refinement it was replaced with a class for just anisotropic refinement. The structure of the application is the same as for the isotropic one, i.e. using a refinement-engine, a mesh-cutter and a refinement-tree. Figure 3.1(b) shows the collaboration diagram of the application and Figure 3.1(a) shows a high level flow diagram of the application. From these figures we can see how the refinement-engine, mesh-cutter and refinement-tree collaborate.

To be able to refine a cell some scheme for orienting the cell is needed, without an orientation it is impossible to know how to execute a split in the wanted direction. The created scheme makes use of corner points as its underlying framework where each cell has a list of its corner points. The corner points are used to make sure that a split is done properly by splitting between predetermined pairs of corner points. The corner points are defined as the corners of a hexahedron which means that every cell will always have eight corner points. When the mesh is created, or rather read, the corner points will need to be calculated. Naturally just knowing the corner points will not tell you between which pair of points a split is done, therefore a uniform way of indexing the corner points is needed. The corner points are indexed according to Figure 3.2(a). Note that the numbering can be arbitrary as long as we keep track of how the split will be done, that is knowing between which pair of points the splits are to be inserted. If we look at figure 3.2 the edge $A \rightarrow D$ will be one axis, the edge $A \rightarrow E$ another axis and the edge between $A \rightarrow B$ the last axis. Because of this the direction of a split is stored relative to the corner points and not the axis.



(a) (b)

Figure 3.1: Figure (a) shows the high level flow graph of the application. For every time-step this flow graph is executed. Figure (b) show the collaboration diagram for the class `dynamicRefineFvMeshHexRef2`.



(a) (b) (c)

Figure 3.2: An example of a cells corner points. Figure (a) shows the index numbering of the corner points used for all cells. Figure (b) show the labels of a cell and figure (c) shows how the cell in (b) is stored in the list of corner points.

3.1 Refinement-engine

Since `dynamicRefineFvMesh` is tailored to work with octasectal refinement some modifications are needed for it to work for anisotropic refinement. The major change is in the selection of cells to refine. Since we use corner-points and split directions for the cells we need a way of selecting in which direction to split. This functionality is implemented in `selectRefine CandidatesHessian` by computing the hessian to find the most effective (according to the hessian) direction to split a cell. Recall that the hessian is the square matrix of the second-order partial derivaties of a function, so it describes how the function curves and bends with respect to its variables. From the hessian we can approximate in which direction the flux fluctuate the most. There is of course indefinitely many ways of choosing the best cells and directions to split and using the hessian is only one such way. The most prominent function in this class is `update` which selects the refinement candidates by calling `selectRefineCandidatesHessian` and then call the mesh-cutter to refine the cells that are possible to refine. One difference is that `dynamicRefineFvMesh` now also handles the redistribution of the mesh. This is achieved by using the `decompositionMethod` class. From this we get a list that for every cell in the mesh contains the processor number of the processor it should be sent to. With this information the mesh is distributed to the other processors. Redistributing the mesh will make the calculated corner points and the refinement-tree invalid since they depend upon cells that may be distributed. To counteract this problem the refinement-tree contains a function to redistribute itself, likewise the mesh-cutter also contains a function to redistribute the corner-points.

3.2 Mesh-cutter

The mesh-cutter had to be rewritten to handle anisotropic refinement. When doing cell splitting we need to make sure the mesh is consistent and satisfies the “two-to-one” rule. If we look at figure 3.3 we can see the three different ways the side of a hexahedron can be split and two cases where splitting is not allowed. Note that these different cases assume the mesh is consistent, any consideration for inconsistent meshes is not taken. From these cases we can see that for a face to be splittable it needs to contain four corner points from its neighbour. The neighbour is in this setting defined as the neighbouring cell to the cell marked for refinement. From this we can formulate a simple constraint for keeping a mesh consistent;

let c be a cell marked for refinement and $F : \{f_0,...,f_n\}$ the set of faces f_i on c needed to be split for a certain anisotropic refinement direction d of c. For every face $f \in F$ we define the set N of neighbour corner points:

$$N(f) = \begin{cases} \text{Corner points of f:s face neighbour} & \text{if neighbour exist} \\ \emptyset & \text{otherwise} \end{cases}$$

We say that a cell is consistently refinable if all faces with a neighbouring cell contain exactly four corner points on it or the neighbour is nonexistent. More formally, c is consistently refinable

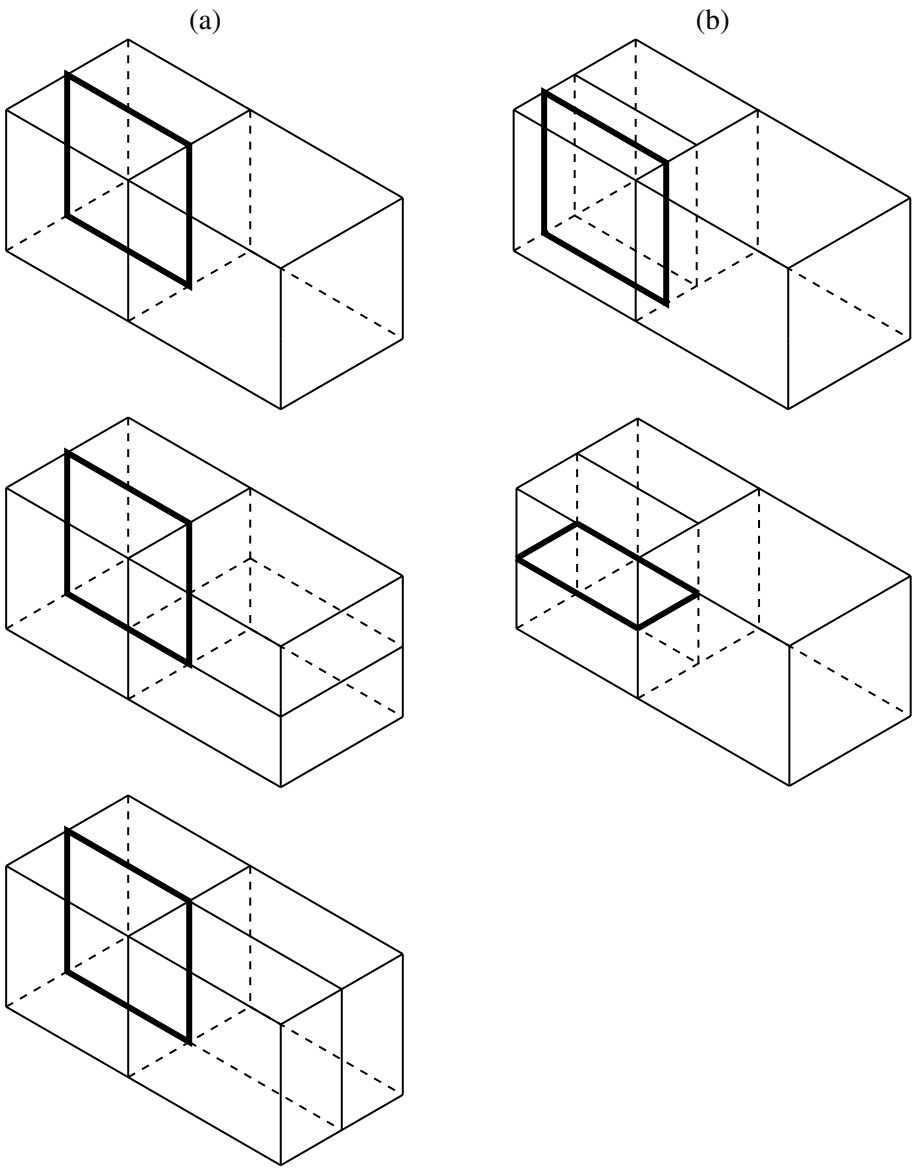


Figure 3.3: Column (a) show the three different types of splits a side of a hexahedron is allowed to do and column (b) shows the disallowed splits. The bold lines represent the edges the cell split aims to create.

in direction d if:

$$\forall f \in F, N(f) = \{\emptyset\} \vee |N(f)| = 4$$

The class `hexRef2` provides the functionality of consistent refinement in the function `consistentRefinement`. The function `consistentRefinement()` takes a list of cell labels and their split direction as input, then unmarks the cells that are not splittable and returns a new list of cells to split.

The actual refinement commands will be executed when calling `setRefinement()`. The function `setRefinement()` takes as input a list of cells for refinement and an object of `polyTopoChange`. The class `polyTopoChange` contains functions for changing the mesh (adding/removing points, faces and cells) and also holds the new mesh that is updated whenever any mesh-changing function is called. The function `setRefinement()` works by first calculating for all faces by which points or edges the face will be split. The faces are then synchronized between the processor boundaries to know how faces on the other processes will be split. Then for the edges that will get split their midpoints are computed and synchronized. Next the faces that are in some way affected by any change are collected. For instance a change can come from a split, new points or a change of owner and neighbour. Now the new cells for the refinement are created. The cell refinement is achieved by creating one new cell and modifying the old. Now the faces gets split, new points gets added and the neighbours and owners for every face gets updated. Lastly the middle face between the old and new cell is created. For each modified and added cell the corner points gets updated and lastly the split is added to the refinement-tree.

3.3 Refinement-tree

The refinement-tree is replaced with a class `refinementTree` that is based on REFTREE [13]. To limit the amount of work needed to replace `refinementHistory` the public function interface is kept as similar as possible in `refinementTree`. Mitchell describe a way of distributing the refinement-tree across multiple processes where every child can be sent to any process leaving its siblings on another process. This is done by keeping a lightweight substitute data structure of the sibling on the processor it was moved from. This feature was not implemented due to time limit and complexity, however the problem of distribution is solved by another method. This method forces the distribution-method to only distribute children of the root node and with them all their children. It is then up to the application computing the distribution to put the weights on the initial mesh based on the size of the tree for these cells i.e. their number of children. Then the application will compute a load balancing and call `refinementTree` to distribute the tree according to the load balancing. When redistributing OpenFOAM's current refinement history some refined cells might be placed on a different node than some of its sibling cells. This renders these cells unrefineable, not in theory but at least in the current implementation. By using the agglomeration approach in the refinement-tree the phenomenon of orphan cells is avoided completely since only complete family trees are redistributed.

The whole tree is defined by the class `refinementTree` containing a list of children of type `refinementNode`. The `refinementTree` can be seen as the root node and its children as the initial mesh. The class `refinementNode` is a regular node in the tree that contains a variable-sized list of children of type `refinementNode`, see figure 3.4.

It is common in OpenFOAM to do a run, save the case to disk and later continue running the case. This means that we need to be able to read and write the refinement-tree in plaintext.

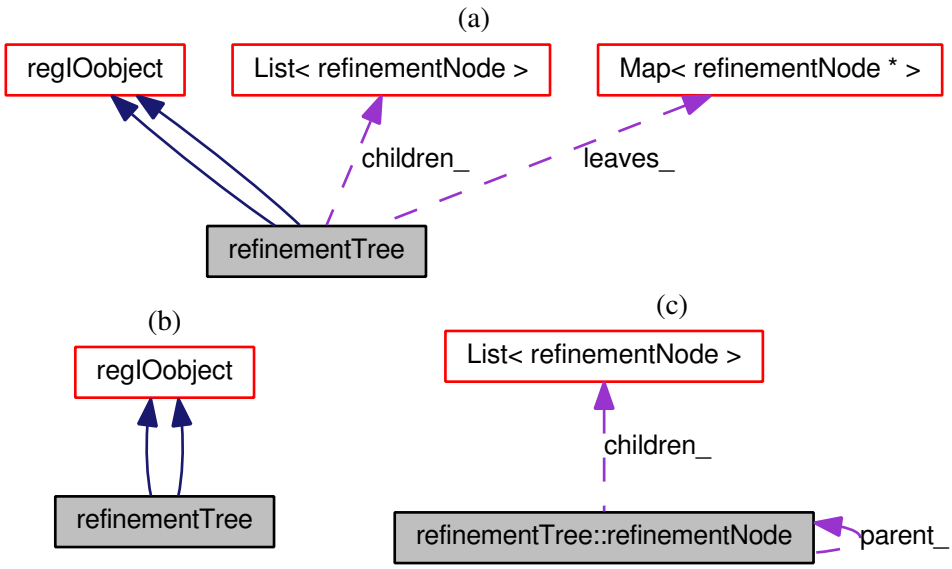


Figure 3.4: Figure (a) and (b) shows the collaboration and inheritance diagram for `refinementTree` and (c) shows the collaboration diagram for `refinementNode`.

Therefore we make `refinementTree` an instance of `regIOobject` which is a class supporting reading and writing itself in plaintext. Being an instance of `regIOobject` means we need to implement functions to read and write the class. To do the conversion from the tree to plaintext we first convert the tree into a list of a list of labels and then let the built-in functions in OpenFOAM for reading and writing this label list handle the rest. The algorithm to transform a tree into a list of list of labels is shown in Algorithm 1 and Figure 3.5 shows an example of its result. Cells that are not visible gets the label -1 since they are not actually present in the mesh. The labels of the leaves are in practice numbers but in the examples letters are used for clarity.

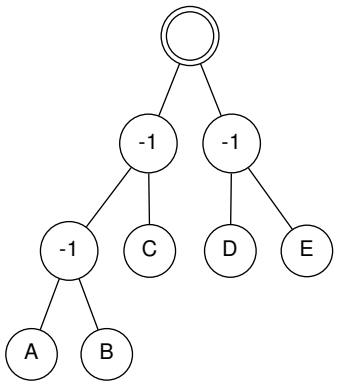
Algorithm 1: TreeToList, converts a refinement-tree into a list of list of labels.

```
Input: A Refinement-tree
Output: The tree as list of list of labels
begin
    create TreePlainList;
    foreach child in Refinement-tree's children do
        ChildList ← NodeToList(child);
        append ChildList to TreePlainList;
    end
    return TreePlainList
end
```

If we look at the function interface for the class `refinementTree` we see that the children of the root node are stored as a list of `refinementNode` in the variable `children_`. The

Algorithm 2: NodeToList converts a refinement-node into a list of labels.

```
Input: A Refinement-node
Output: The refinement-node as a list of labels
begin
    create LabelList;
    append node's label to LabelList;
    append node's number of children to LabelList;
    foreach child of the refinement-node do
        ChildList ← NodeToList(child);
        append ChildList to LabelList;
    end
    return LabelList
end
```



(a)

((-1, 2, -1, 2, A, 0, B, 0, C, 0), (-1, 2, D, 0, E, 0))

(b)

Figure 3.5: Figure (a) shows an example of a refinement-tree and figure (b) shows the tree as its representation in plaintext.

class also contains a variable `leaves_` that is a list of pointers to the leaves in the tree. The refinement-tree provide a function `addSplit()` for adding a split. It takes as input the cell to split and a list of the cell labels for the children. There is also a function `combineCells()` for combining a list of cells into one master cell. The efficiency of updating the tree is dependent upon the efficiency of both the list of leaves and the creation and deletion of the refinementNode objects.

If we look at the functions for adding a refinement to the tree and for combining cells in the tree, we can get an approximation of its complexity. We let l be the leaf that should be split and c_n be the number of children it is split into. Figure 3.6 shows the average and worst-case complexity for the tree. From the figure we see that the average and worst complexity is $\mathrm{O}(c_n)$. Since we know that every split is always into either two, four or eight sub-cells that means that we can interpret c_n as a constant and the complexity of the tree for splitting and combining will

Split Cells:

Description	Average comp.	Worst-case comp.
Find leaf l to add split to (hash-table lookup)	$O(1)$	$O(1)$
Create and copy the list of children to l	$O(c_n)$	$O(c_n)$
Erase l from the list of leaves	$O(1)$	$O(1)$
Set cell label and parent label for every child of l	$O(c_n)$	$O(c_n)$
	$O(c_n)$	$O(c_n)$

Combine Cells:

Description	Average comp.	Worst-case comp.
Erase all the children from the hash-table	$O(c_n)$	$O(c_n)$
Erase all the children from the parent l	$O(c_n)$	$O(c_n)$
	$O(c_n)$	$O(c_n)$

Figure 3.6: The complexity for the refinement-tree when splitting and combining cells.

thus become $O(1)$. The reason we can store the leaves as a list is because the cell labels in OpenFOAM go from zero to the number of cells in the mesh. Naturally these labels are used as indices for the list. When we try to add a label that is out of bounds for the list it gets expanded.

Chapter 4

Test cases

4.1 Refinement-tree

In this section we will compare how well the refinement-tree works in a typical setting, which in this case is in the old mesh-cutter `hexRef8`. We will run a comparison on the wall-clock time on one of OpenFOAM's standard tutorial cases. The case is called breaking of a dam and is a two-dimensional case shown in figure 4.1. The case consists of a column of water resting at the left of the tank. A very thin membrane is holding the water and at time zero the membrane is removed and the water will fall.

We will run the case with one of OpenFOAM's meshcutters, namely `hexRef8` , with its standard refinement history and one run with the same meshcutter but with the refinement-tree described in this thesis. The case is decomposed into two meshes with thirty-two thousand cells and runs from time 0 to 0.4. For the first run we test how well the mesh-cutter `hexRef8` performs with its standard history (RefinementHistory in Table 4.1) and in the second case we test the performance of `hexRef8` with the refinement-tree. During the simulation both refinement and unrefinement occurs which means that the refinement-tree will do both insertions and deletions. Note that this test is not a benchmark but only a crude way of making sure the refinement-tree behaves well in a typical setting.

If we look at the time for the two cases we see that the case with the old refinement history outperforms the refinement-tree case with 1%.

	Execution time	Clock time
Refinement-tree	25911.2 s	25920 s
RefinementHistory	25537.3 s	25560 s

Table 4.1: The execution- and clock time for the dam-break case using hexRef8 and OpenFOAM's refinement history and the refinement-tree respectively.

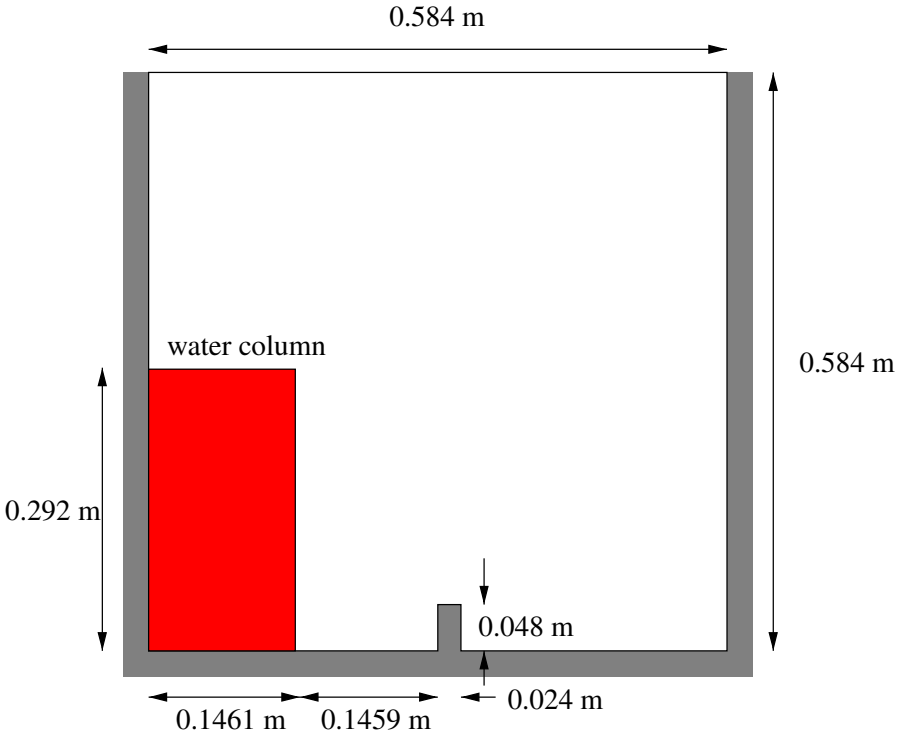


Figure 4.1: The geometry of the dam in the 'breaking of a dam' case.

4.2 CFD Cases

For readers not very familiar with CFD much of the terminology in this chapter will not be comprehendible, but these underlying schemes, methods and values are present for completeness. While the AMR implementation described above is general and not equation or solver dependent, we will here initially only use the steady-state incompressible RANS solver `simpleFoam`, a segregated solver using the SIMPLE algorithm. As the AMR is encapsulated in the OpenFOAM libraries that are linked to the solvers, the solvers themselves needs only minor alteration to incorporate AMR – typically only a few lines of code needs to be changed.

OpenFOAM supports a wide array of discretization schemes but we restrict ourselves to use only one convection scheme: `limitedLinear` which is a second-order TVD scheme. All other operators, the gradient, the Laplacian, interpolation, etc, are all employing second-order central differencing.

With regard to AMR we use a very simple error indicator, based on the jump in pressure over the cell faces see e.g. [15, 16], for both isotropic and anisotropic mesh refinement. For the anisotropic refinement we use the Hessian of pressure to determine the direction of the split. Please note that a present restriction of the AMR implementation is that we do not allow any

refinement of wall boundary cells, the y^+ is set manually and the cells are not to be modified during the simulations. In these simulations the anisotropic refinements differ in its refining method compared to the isotropic one since it is not using a method called *spreading*. Imagine a scenario where a cell needs to be refined but is constrained, due to 2:1 consistency, by its neighbours geometry. In this scenario the application should be able to recursively refine the neighbouring cells until the cell becomes refinable, which is the method called spreading.

4.2.1 Backward facing step

The classical backward facing step in the ERCOFTAC C-30 database is initially used for testing the AMR. The Reynolds number based on step height, H m is 50000 and the tunnel extends is $40H$ downstream of the step and $4H$ upstream. The height of the tunnel is $9H$. We use the Spalart-Allmaras RANS model with a continuous wall function and a y^+ of roughly 10. The inflow conditions are the same as developed for the Lisbon Uncertainty Workshops.

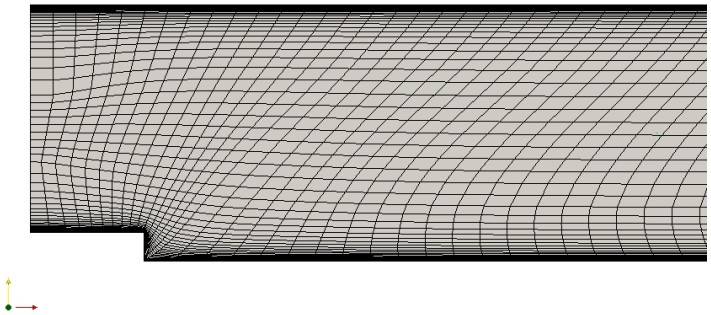
We use the pressure jump error indicator with a tolerance of 1×10^{-3}. Figure 4.2 shows the resulting meshes using anisotropic and isotropic refinement, respectively. The initial mesh was made up of 3300 hexahedra (as computations in OpenFOAM always is 3D). The used tolerance is rather lax and the isotropic refinement needs just one refinement step to fulfill the tolerance, ending up with a mesh of 5000 cells while the anisotropic mesh is just short of 4000 cells. Both simulations show similar results with a reattachment length of roughly 6.

4.2.2 NACA0012 wing

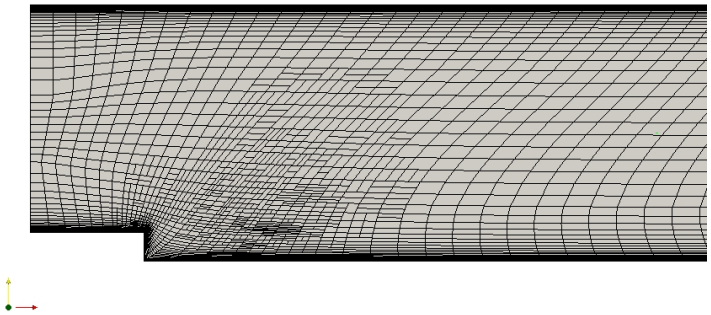
Next we apply the AMR to resolve the tip-vortex of a NACA0012 with a rounded wing tip at 10 degrees angle of attack. The chord based Reynolds number is 4.35×10^6. We compute this with anisotropic and isotropic refinement using a tolerance of the error indicator of 1×10^{-4}. Here we also turn off the spreading in isotropic refinement.

In Figure 4.3(a) the refinement is shown. The bulk of the refinement occurs in the downstream tip vortex and some at the leading edge of the wing. The areas are virtually identical for the three AMR approaches but the isotropically refined mesh is denser (0.90M cells) compared to isotropic with no spreading (0.89M) and anisotropic (0.78M). The pressure in the vortex core (defined as the position of minimum pressure) is presented in Figure 4.3b. Here we also compare to two static meshes (gridA has 0.75M and gridC 2.98M cells). It is interesting to note that it is only the standard isotropic AMR that picks up the low pressure at $x/c \approx 0.7$, indicating that the only a few restrictions of cells that should be refined can be devastating. We also believe that the lack of spreading gives a lot of unnecessary hanging nodes that are responsible for the rapid damping of the core pressure (as this does not happen for the static coarse mesh) .

(a)



(b)



(c)

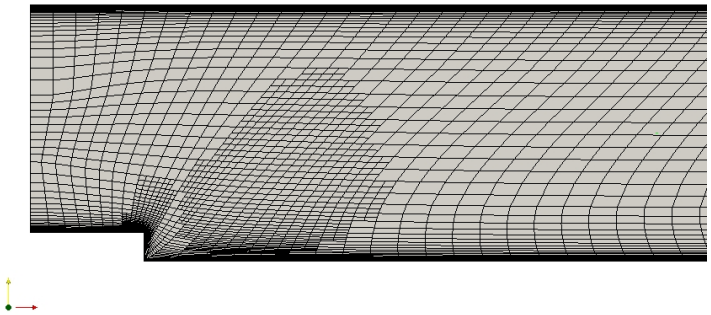


Figure 4.2: Meshes for the backward facing step: (a) initial mesh, (b) anisotropic mesh and (c) isotropic mesh.

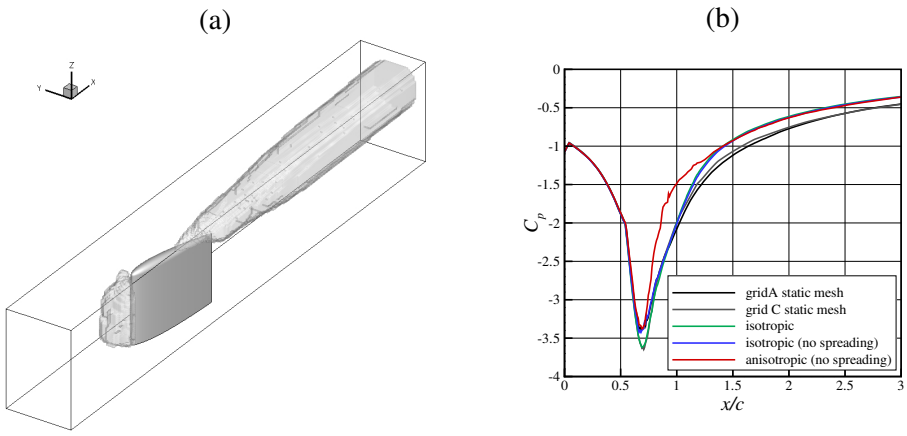


Figure 4.3: NACA0012 wing. (a) areas of refinement and (b) pressure in the vortex core.

Chapter 5

Conclusions

From the design criteria set up in chapter 1 we saw that an application with the following functionality was needed:

- Select cells to refine and the direction in which to split them.
- Split selected cells anisotropically.
- Store the refinement changes in a refinement-tree.
- Distribute the refinement-tree.
- Automatically refine and redistribute the mesh and tree.

Also some demands on the refinement-tree were set so it should satisfy these criteria:

- Handle any type of cell (octahedra and hexahedra among others).
- Function in parallel and handle distribution between processes.
- Efficient in terms of memory and computation cost.

The result of this thesis show that all these items were indeed possible to achieve although some limitations are set; like only allowing consistent meshes. If one were to allow inconsistent meshes more work on the orientation of the cells would be needed since the cells could appear in unexpected geometries. By looking at the cases computed in the previous section we can see that the implementation described achieves its goal of doing anisotropic AMR. Since this thesis introduces an abstract (not bound to any programming language) scheme for doing AMR some interested parties could, by using the same or a similar scheme, implement it in other settings. Ultimately the aim is that this particular application will reach its feature completion and be part of the standard OpenFOAM distribution. This thesis also argue that using a basic refinement-tree together with the AMR enables it to do unrefinement, altough this is not shown or proven.

5.1 Future work

What has been described in this thesis is a primitive form of anisotropic AMR. It is primitive because it lacks some features that are almost vital for the application to be workable in real-life scenarios. The implementation lacks two important features; spreading and unrefinement. The unrefinement is a very crucial component particularly for cases where some point of interest moves across the domain during the simulation. Not using unrefinement in cases like that will result in a mesh that is too fine-coarsed in some areas of the domain. Achieving a too fine-coarsed grid from AMR defeats one of its main goals which is reducing the number of cells in the mesh. Implementing unrefinement for a split into two should be quite straightforward. For instance you could use the refinement-tree to find siblings and then merge them by removing the face between the cells and then coalesce the points. Another useful improvement would be on the initial selection of corner points. In the current state corner points can only be created from perfect hexahedra thus forcing that no side of the cell is split beforehand. This becomes a problem when for instance an anisotropically refined mesh is exported and the corner points are lost then the corner points cannot be calculated for several cells since they are not proper hexahedra. So creating a more competent function for calculating the corner points would be quite beneficial too. Some improvements on the refinement-tree could also be done, in particular adding features desribed in [13] like unrestrictive redistribution and using space filling curves as a partitioning method.

Bibliography

[1] E. F. Toro, Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction, Springer, 1999.

[2] OpenFOAM - The Open Source Computational Fluid Dynamics (CFD) Toolbox (June 2012).
URL http://openfoam.com/

[3] E. Süli, An introduction to numerical analysis, Cambridge University Press, Cambridge New York, 2003.

[4] Handbook of grid generation, CRC Press, Boca Raton, Fla, 1999.

[5] H. Jasak, Error analysis and estimation in the Finite Volume method with applications to fluid flows, Ph.D. thesis, Imperial College, University of London (1996).

[6] J.-F. Remacle, S. S. Frazão, X. Li, M. S. Shephard, An adaptive discretization of shallow-water equations based on discontinuous Galerkin methods, International Journal for Numerical Methods in Fluids 52 (8) (2006) 903–923.
URL http://dx.doi.org/10.1002/fld.1204

[7] L. Demkowicz, J. Oden, W. Rachowicz, O. Hardy, Toward a universal h-p adaptive finite element strategy, part 1. Constrained approximation and data structure, Computer Methods in Applied Mechanics and Engineering 77 (1–2) (1989) 79 – 112.

[8] R. B. Andrew, A. H. Sherman, A. Weiser, Some Refinement Algorithms and Data Structures For Regular Local Mesh Refinement (1983).

[9] I. Foster, Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, Addison Wesley, 1995.
URL http://www.mcs.anl.gov/~itf/dbpp/text/node94.html

[10] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering (2012).
URL http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[11] SCOTCH – Software package and libraries for sequential and parallel graph partitioning, static mapping, and sparse matrix block ordering, and sequential mesh and hypergraph partitioning (2012).
URL http://www.labri.fr/perso/pelegrin/scotch/

[12] Q. P. Hafermann D., E. C., Parallelisation and load balancing in dynamic grid control. Confidential Report, Streamline Deliverable 32.2. (2012).

[13] W. F. Mitchell, A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids, J. Parallel Distrib. Comput. 67 (4) (2007) 417–429.
URL http://dx.doi.org/10.1016/j.jpdc.2006.11.003

[14] OpenFOAM Foundation, OpenFOAM User Guide, version 2.1.1 (May 2012).
URL http://foam.sourceforge.net/docs/Guides-a4/UserGuide.pdf

[15] C. Eskilsson, R. Bensow, A mesh adaptive compressible Euler model for the simulation of cavitating flow, Proceedings of the IV International Conference on Computational Methods in Marine Engineering (2011) 458–469.

[16] J. Windt, C. Klaij, Adaptive mesh refinement in MARIN's viscous flow solver ReFRESCO: Implementation and application to steady flow, Proceedings of the IV International Conference on Computational Methods in Marine Engineering (2011) 528–543.

[17] OpenFOAM C++ Documentation (Novemeber 2012).
URL http://www.openfoam.org/docs/cpp/

Appendix A

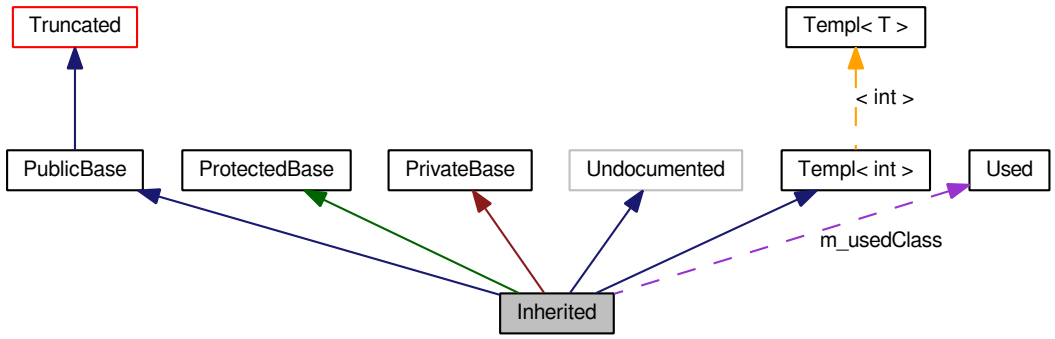
Graph legend

This page explains how to interpret the graphs that are generated by Doxygen [17]. Consider the following example:

```
/*! Invisible class because of truncation */
class Invisible { };

/*! Truncated class, inheritance relation is hidden */
class Truncated : public Invisible { };

/* Class not documented with doxygen comments */
class Undocumented { };

/*! Class that is inherited using public inheritance */
class PublicBase : public Truncated { };

/*! A template class */
template<class T> class Templ { };

/*! Class that is inherited using protected inheritance */
class ProtectedBase { };

/*! Class that is inherited using private inheritance */
class PrivateBase { };

/*! Class that is used by the Inherited class */
class Used { };

/*! Super class that inherits a number of other classes */
class Inherited : public PublicBase,
                  protected ProtectedBase,
                  private PrivateBase,
                  public Undocumented,
```

```
                    public Templ<int>
{
  private:
    Used *m_usedClass;
};
```

This will result in the following graph:



The boxes in the above graph have the following meaning:

- A filled gray box represents the struct or class for which the graph is generated.
- A box with a black border denotes a documented struct or class.
- A box with a grey border denotes an undocumented struct or class.
- A box with a red border denotes a documented struct or class forwhich not all inheritance/containment relations are shown. A graph is truncated if it does not fit within the specified boundaries.

The arrows have the following meaning:

- A dark blue arrow is used to visualize a public inheritance relation between two classes.
- A dark green arrow is used for protected inheritance.
- A dark red arrow is used for private inheritance.
- A purple dashed arrow is used if a class is contained or used by another class. The arrow is labeled with the variable(s) through which the pointed class or struct is accessible.
- A yellow dashed arrow denotes a relation between a template instance and the template class it was instantiated from. The arrow is labeled with the template parameters of the instance.

Appendix B

Code documentation

B.0.1 dynamicRefineFvMeshHexRef2 Class Reference

A fvMesh with built-in refinement. Refines Hexahedra into two subcells. Determines which cells to refine/unrefine and does all in **update()**.

Public Member Functions

- **TypeName** ("dynamicRefineFvMeshHexRef2")
- **dynamicRefineFvMeshHexRef2** (const IOobject &io)
- virtual ∼**dynamicRefineFvMeshHexRef2** ()
- const **hexRef2** & **meshCutter** () const
- const PackedBoolList & **protectedCell** () const
- virtual bool **update** ()
- virtual bool **writeObject** (IOstream::streamFormat fmt, IOstream::versionNumber ver, IOstream::compressionType cmp) const

Protected Member Functions

- void **readDict** ()
- autoPtr< mapPolyMesh > **refine** (const labelList &cellsToRefine, const labelList &splitCellDirection)
- virtual void **selectRefineCandidatesHessian** (const scalar lowerRefineLevel, const scalar upperRefineLevel, const volScalarField &vFld, PackedBoolList &candidateCell, PackedList< 2 > &splitCellDirection, scalar maxAspectRatio)
- virtual Pair< labelList > **selectRefineCells** (const label maxCells, const label maxRefinement, const PackedBoolList &candidateCell, PackedList< 2 > &splitCellDirection)
- scalar **getMergeDistance** (const Time &runTime, const boundBox &bb)

Static Protected Member Functions

- static label **count** (const PackedBoolList &, const unsigned int)

Protected Attributes

- **hexRef2 meshCutter_**
- HashTable< word > **correctFluxes_**
- label **nRefinementIterations_**
- PackedBoolList **protectedCell_**

Private Member Functions

- **dynamicRefineFvMeshHexRef2** (const **dynamicRefineFvMeshHexRef2** &)
- void **operator=** (const **dynamicRefineFvMeshHexRef2** &)

B.0.1.1 Detailed Description

Source files

- `dynamicRefineFvMeshHexRef2.H`
- `dynamicRefineFvMeshHexRef2.C`

B.0.1.2 Constructor & Destructor Documentation

dynamicRefineFvMeshHexRef2 (const dynamicRefineFvMeshHexRef2 &) `[private]`

Disallow default bitwise copy construct.

dynamicRefineFvMeshHexRef2 (const IOobject & *io*) `[explicit]`

Construct from IOobject.

~dynamicRefineFvMeshHexRef2 ( ) `[virtual]`

Destructor.

B.0.1.3 Member Function Documentation

Foam::label count (const PackedBoolList & *l*, const unsigned int *val*) `[static, protected]`

Count set/unset elements in packedlist.

void readDict ( ) `[protected]`

Read the projection parameters from dictionary.

Foam::autoPtr< Foam::mapPolyMesh > refine (const labelList & *cellsToRefine,* const labelList & *splitCellDirection*) `[protected]`

Refine cells in given direction. Update mesh and fields.

void selectRefineCandidatesHessian (const scalar *lowerRefineLevel,* const scalar *upperRefineLevel,* const volScalarField & *vFld,* PackedBoolList & *candidateCell,* PackedList< 2 > & *splitCellDirection,* scalar *maxAspectRatio*) `[protected, virtual]`

Select candidates for refinement based on the hessian.

and set the candidates split direction

Foam::Pair< Foam::labelList > selectRefineCells (const label *maxCells,* const label *maxRefinement,* const PackedBoolList & *candidateCell,* PackedList< 2 > & *splitCellDirection*) `[protected, virtual]`

Subset candidate cells for refinement.

scalar getMergeDistance (const Time & *runTime,* const boundBox & *bb*) `[protected]`

Get the merge distance for distribution.

void operator= (const dynamicRefineFvMeshHexRef2 &) `[private]`

Disallow default bitwise assignment.

TypeName ("dynamicRefineFvMeshHexRef2")

Runtime type information.

const hexRef2& meshCutter ( ) const `[inline]`

Direct access to the refinement engine.

const PackedBoolList& protectedCell ( ) const `[inline]`

Cells which should not be refined/unrefined.

bool update ( ) [virtual]

Update the mesh for both mesh motion and topology change.

Implements **dynamicFvMesh** .

bool writeObject (IOstream::streamFormat *fmt,* IOstream::versionNumber *ver,* IOstream::compressionType *cmp*) const [virtual]

Write using given format, version and compression.

B.0.1.4 Member Data Documentation

hexRef2 meshCutter_ [protected]

Mesh cutting engine.

HashTable<word> correctFluxes_ [protected]

Fluxes to map.

label nRefinementIterations_ [protected]

Number of refinement/unrefinement steps done so far.

PackedBoolList protectedCell_ [protected]

Protected cells (usually since not hexes)

B.0.2 hexRef2 Class Reference

Anisotropic refinement of hexahedrons using **polyTopoChange**.

Public Member Functions

- **ClassName** ("hexRef2")
- **hexRef2** (const polyMesh &mesh)
- **hexRef2** (const polyMesh &mesh, const **refinementTree** &refTree)
- const polyMesh & **mesh** () const
- const **refinementTree** & **refTree** () const
- const labelListIOList & **cornerPoints** ()
- Pair< Foam::labelList > **consistentRefinement** (const labelList &cellsToRefine, const PackedList< 2 > &splitDirection)
- labelList **setRefinement** (const labelList &cellLabels, const labelList &splitDirection-Packed, **polyTopoChange** &meshMod)
- void **updateMesh** (const mapPolyMesh &)
- void **updateMesh** (const mapPolyMesh &, const Map< label > &pointsToRestore, const Map< label > &facesToRestore, const Map< label > &cellsToRestore)
- void **subset** (const labelList &pointMap, const labelList &faceMap, const labelList &cellMap)
- void **distribute** (const mapDistributePolyMesh &)
- const labelList & **cellLevel** ()
- void **checkMesh** () const
- void **setInstance** (const fileName &inst)
- bool **write** () const

Private Member Functions

- point **calculateFaceMidPoint** (const label faceLabel) const
- void **calculateCornerPoints** ()
- void **calculateSplittableFaces** (const labelListList &cornerPoints, boolList &splittable-Faces) const
- void **unmarkInconsistentCells** (const labelList &cellsToRefine, PackedBoolList &**re-fineCell**, boolList &splittableFaces, const PackedList< 2 > &splitDirection) const
- void **orientCornerPoints** (const labelList &cornerPointsI, labelList &tempCornerPoints, const label splitDirection) const
- int **set_intersection_size** (const labelList &a, const labelList &b) const
- void **addCornerPoints** (const labelList &cellAddedCell, const labelList &splitDirection, const labelList &edgeMidPoint)
- void **arrangeFace** (labelList &faceVerts, label faceI)

- label **middlePointGeometric** (const label point0, const label point1, const label cellI) const
- label **pointIndexFace** (const label pointLabel, const label faceLabel)
- label **findInternalFacePoint** (const labelList &pointLabels) const
- label **basePointFace** (const label pA, const label pB, const label flabel)
- bool **onBaseCell** (const labelList &splitDirection, const label cellI, const label pointI) const
- label **faceFromPoints** (const label cellLabel, const labelList &points) const
- void **getFaceInfo** (const label faceI, label &patchID, label &zoneID, label &zoneFlip) const
- label **addFace** (**polyTopoChange** &meshMod, const label faceI, const face &newFace, const label own, const label nei) const
- void **modFace** (**polyTopoChange** &meshMod, const label faceI, const face &newFace, const label own, const label nei) const
- void **getFaceNeighbours** (const labelList &cellAddedCell, const labelList &splitDirection, const label faceI, const label pointI, label &own, label &nei) const
- void **dumpCell** (const label cellI) const
- **hexRef2** (const **hexRef2** &)
- void **operator=** (const **hexRef2** &)

Private Attributes

- const polyMesh & **mesh_**
- labelListIOList **cornerPoints_**
- labelIOList **cellLevel_**
- **refinementTree refTree_**
- **removeFaces faceRemover_**

B.0.2.1 Detailed Description

Source files

- `hexRef2.H` hexRef2C

B.0.2.2 Constructor & Destructor Documentation

hexRef2 (const hexRef2 &) `[private]`

Disallow default bitwise copy construct.

hexRef2 (const polyMesh & *mesh*)

Construct from mesh, read_if_present refinement data and corner points (from write below)

hexRef2 (const polyMesh & *mesh,* const refinementTree & *refTree*)

Construct from mesh and refinement-tree.

B.0.2.3 Member Function Documentation

Foam::point calculateFaceMidPoint (const label *faceLabel*) const `[private]`

Calculate the intersection point of the face.

void calculateCornerPoints ( ) `[private]`

Calculates the corner points for every cell.

void calculateSplittableFaces (const labelListList & *cornerPoints,* boolList & *splittableFaces*) const `[private]`

Calculates for all faces if they are splittable A face is splittable if it contains four corner points from its owner and neighbour.

Parameters

cornerPoints	The Corner points for all cells
splittable-Faces	The list showing if a face is split

void unmarkInconsistentCells (const labelList & *cellsToRefine,* PackedBoolList & *refineCell,* boolList & *splittableFaces,* const PackedList< 2 > & *splitDirection*) const `[private]`

Unmarks cells that cannot be split in the given direction.

Parameters

cellsToRefine	The cells we want to refine
refineCell	True if cell can and will be refined, false otherwise
splittable-Faces	True if face is splittable, false otherwise
splitDirection	The direction we want to split the cells

void orientCornerPoints (const labelList & *cornerPointsl,* labelList & *tempCornerPoints,* const label *splitDirection*) const `[private]`

Orients a cells corner points according to the split direction.

Parameters

cornerPointsI	The corner points for the cell
tempCorner-Points	The oriented corner points
splitDirection	The split direction for the cell

int set_intersection_size (const labelList & *a,* const labelList & *b*) const `[private]`

Returns the size of the set intersection of two label lists.

void addCornerPoints (const labelList & *cellAddedCell,* const labelList & *splitDirection,* const labelList & *edgeMidPoint*) `[private]`

Adds corner points for the split cells (overwrites cornerPoints_)

Parameters

cellAddedCell	For each refined cell the added cell's label
splitDirection	For each refined cell the direction it is split
edgeMidPoint	For all edges if the edge is split edgeMidPoint contains the label of the added point on the edge

void arrangeFace (labelList & *faceVerts,* label *faceI*) `[inline, private]`

Arranges the face so that a corner point is first in the list.

Parameters

faceVerts	The list of points making the face
faceI	Label of original face faceVerts was created from

Foam::label middlePointGeometric (const label *point0,* const label *point1,* const label *cellI*) const `[private]`

Returns the point geometrically between two points in a Cell. If no such point exists returns -1.

Parameters

point0	The first point
point1	The second point
cellI	The cell containing the points

Foam::label pointIndexFace (const label *pointLabel,* const label *faceLabel*) `[private]`

Returns the index of a point in a face. For instance if the point is first in the list the value 0 is returned.

Parameters

pointLabel	The label of the point
faceLabel	The label of the face

Foam::label findInternalFacePoint (const labelList & *pointLabels*) const `[private]`

Returns first point in pointLabels that uses an internal face. Used to find point to inflate cell/face from (has to be connected to internal face). Returns -1 (so inflate from nothing) if none found.

Foam::label basePointFace (const label *pA,* const label *pB,* const label *flabel*) `[private]`

Goes through a face and returns the index of the point such that it is _either_ pA and next point in the face is pB _or_ it is pB and next point is pA.

Parameters

pA	The first point
pB	The second point
flabel	The label of the face

bool onBaseCell (const labelList & *splitDirection,* const label *cellI,* const label *pointI*) const `[private]`

Returns true if the point is on the base of the cell and false if it is on the extended (refined) cell.

Parameters

splitDirection	The split direction for all cells
cellI	The that is split
pointI	The global point we want to check

Foam::label faceFromPoints (const label *cellLabel,* const labelList & *points*) const `[private]`

Return the face containing the given points on the cell. Note that the points can be a subset of the face.

Parameters

cellLabel	The label of the cell we search
points	The points comprising the face

void getFaceInfo (const label *facel,* label & *patchID,* label & *zoneID,* label & *zoneFlip*) const `[private]`

Get patch and zone info.

Foam::label addFace (polyTopoChange & *meshMod,* const label *facel,* const face & *newFace,* const label *own,* const label *nei*) const `[private]`

Adds a face on top of existing faceI. Reverses if nessecary.

void modFace (polyTopoChange & *meshMod,* const label *facel,* const face & *newFace,* const label *own,* const label *nei*) const `[private]`

Modifies existing faceI for either new owner/neighbour or new face points. Reverses if nessecary.

void getFaceNeighbours (const labelList & *cellAddedCell,* const labelList & *splitDirection,* const label *facel,* const label *pointI,* label & *own,* label & *nei*) const `[private]`

Get new owner and neighbour of pointI on faceI.

void dumpCell (const label *celll*) const `[private]`

Debugging: dump cell as .obj file.

void operator= (const hexRef2 &) `[private]`

Disallow default bitwise assignment.

ClassName ("hexRef2")

Runtime type information.

const polyMesh& mesh ( ) const `[inline]`

Returns the mesh.

const refinementTree& refTree ( ) const `[inline]`

Returns the refinement tree.

const labelListIOList& cornerPoints ( ) `[inline]`

Returns the corner points.

Foam::Pair< Foam::labelList > consistentRefinement (const labelList & *cellsToRefine,* const PackedList< 2 > & *splitDirection*)

Given valid mesh and proposed cells to refine calculate any clashes (due to 2:1) and return ok list of cells to refine and their split direction. Will remove cells not refinable from the set. Returns a pair of label lists where first list is the labels of the cells to refine and second list is their direction.

Parameters

cellsToRefine	The cells we want to refine
splitDirection	The direction we want to refine

Foam::labelList setRefinement (const labelList & *cellLabels,* const labelList & *splitDirection-Packed,* polyTopoChange & *meshMod*)

Insert refinement. All selected cells will be split into 2. Returns per element in cellLabels the cell it was split into.

void updateMesh (const mapPolyMesh & *map*)

Update local numbering for changed mesh.

void updateMesh (const mapPolyMesh & *map,* const Map< label > & *pointsToRestore,* const Map< label > & *facesToRestore,* const Map< label > & *cellsToRestore*)

Update local numbering.

void subset (const labelList & *pointMap,* const labelList & *faceMap,* const labelList & *cellMap*)

Update local numbering for subsetted mesh. Gets new-to-old maps. Not compatible with unrefinement.

void distribute (const mapDistributePolyMesh & *map*)

Update local numbering for mesh redistribution.

const labelList& cellLevel ( ) [inline]

Return cell level.

void checkMesh ( ) const

Debug: Check coupled mesh for correctness.

void setInstance (const fileName & *inst*)

Set instance for mesh files.

bool write ( ) const

Force writing refinement tree to polyMesh directory.

B.0.2.4 Member Data Documentation

const polyMesh& mesh_ [private]

Reference to underlying mesh.

labelListIOList cornerPoints_ [private]

Per cell its corner points.

labelIOList cellLevel_ [private]

Per cell the refinement level.

refinementTree refTree_ [private]

Refinement history.

removeFaces faceRemover_ [private]

Face remover engine.

B.0.3 refinementTree Class Reference

The complete **refinementTree**.

Classes

- class **refinementNode**

Public Member Functions

- **refinementTree** (const IOobject &)
- **refinementTree** (const IOobject &, const label)
- **refinementTree** (const IOobject &, const **refinementTree** &)
- **refinementTree** (const IOobject &, Istream &)
- bool **active** () const
- void **addSplit** (const label cellLabel, const labelList &children)
- void **addSplit** (const label cellLabel, const label siblingLabel)
- void **combineCells** (const label masterCellI, const labelList &combinedCells)
- **refinementNode** ∗ **findLeaf** (const label cellLabel) const
- label **sameProc** (const **refinementNode** &child, const labelList &toProc)
- void **calculateAgglomeration** (const polyMesh &mesh, labelList &fineToCoarse, vectorField &coarsePoints, scalarField &coarseWeights) const
- const **refinementNode** ∗ **parent** (label cellLabel) const
- const labelList **leavesList** () const
- label **nLeaves** () const
- label **nChildren** () const
- label **getDepth** (const label cellLabel)
- void **updateMesh** (const mapPolyMesh &)
- void **subset** (const labelList &pointMap, const labelList &faceMap, const labelList &cellMap)
- void **distribute** (const mapDistributePolyMesh &)
- void **writeDebug** () const
- virtual bool **readData** (Istream &)
- virtual bool **writeData** (Ostream &) const
- bool **operator==** (const **refinementTree** &rt) const
- bool **operator!=** (const **refinementTree** &rt) const

Private Member Functions

- **TypeName** ("refinementTree")
- **refinementNode** ∗ **findNode** (const label cellLabel)
- label **numChildren** () const
- label **firstLeaf** (const **refinementNode** &child)
- bool **createMap** ()
- void **fromListList** (const labelListList &childToProc)
- void **addChildrenDistributed** (const labelListList &newChildren, const Map< label > &oldToNew)
- void **calculateFineToCoarse** (const polyMesh &mesh, labelList &fineToCoarse, const List< **refinementNode** > &children, vector &coarsePoint, label coarseCell, int &weight, bool &firstTime, labelList &testList) const

Private Attributes

- List< **refinementNode** > **children_**
- List< **refinementNode** ∗ > **leaves_**

Friends

- Istream & **operator**>> (Istream &, **refinementTree** &)
- Ostream & **operator**<< (Ostream &, const **refinementTree** &)

B.0.3.1 Detailed Description

A **refinementTree** contains a list of refinementNodes, its children.

Source files

- `refinementTree.H`
- `refinementTree.C`

B.0.3.2 Constructor & Destructor Documentation

refinementTree (const IOobject & *io*)

Construct (read) given an IOobject.

refinementTree (const IOobject & *io,* const label *nCells*)

Construct (read) or construct from initial number of cells.

refinementTree (const IOobject & *io,* const refinementTree & *rt*)

Construct as copy.

refinementTree (const IOobject & , Istream &)

Construct from Istream.

B.0.3.3 Member Function Documentation

TypeName ("refinementTree") `[private]`

Foam::refinementTree::refinementNode * findNode (const label *cellLabel*) `[private]`

Find node from label (return NULL if not present)

label numChildren ( ) const `[inline, private]`

Number of children.

Foam::label firstLeaf (const refinementNode & *child*) `[private]`

Find and return the label of the first leaf of the node.

bool createMap ( ) `[private]`

Construct the hashmap.

void fromListList (const labelListList & *childToProc*) `[private]`

Construct tree from labelListList.

void addChildrenDistributed (const labelListList & *newChildren,* const Map< label > & *oldToNew*) `[private]`

Add children from distribution.

void calculateFineToCoarse (const polyMesh & *mesh,* labelList & *fineToCoarse,* const List< refinementNode > & *children,* vector & *coarsePoint,* label *coarseCell,* int & *weight,* bool & *firstTime,* labelList & *testList*) const `[private]`

Calculates which coarse cell a refined cell is part of.

bool active ( ) const `[inline]`

Is there unrefinement history.

void addSplit (const label *cellLabel,* const labelList & *children*)

Add a split to a cell.

void addSplit (const label *cellLabel,* const label *siblingLabel*)

Add a split to a cell with only one sibling.

void combineCells (const label *masterCelll,* const labelList & *combinedCells*)

Store combining cells into master.

refinementNode∗ findLeaf (const label *cellLabel*) const `[inline]`

Look for leaf in leaves_.

Foam::label sameProc (const refinementNode & *child,* const labelList & *toProc*)

Check if the children are in fact sent to the same processor.

void calculateAgglomeration (const polyMesh & *mesh,* labelList & *fineToCoarse,* vectorField & *coarsePoints,* scalarField & *coarseWeights*) const

Calculate agglomeration for (re)distribution.

const refinementNode∗ parent (label *cellLabel*) const `[inline]`

Return parent for cell.

const Foam::labelList leavesList ( ) const

Returns the leaves as a label list.

label nLeaves ( ) const `[inline]`

Returns the number of leaves.

label nChildren ( ) const `[inline]`

Returns the number of the root's immediate children.

Foam::label getDepth (const label *cellLabel*)

Returns the depth in the tree of a cell.

void updateMesh (const mapPolyMesh & *map*)

Update numbering for mesh changes.

void subset (const labelList & *pointMap,* const labelList & *faceMap,* const labelList & *cellMap*)

Update numbering for subsetting.

void distribute (const mapDistributePolyMesh & *map*)

Update local numbering for mesh redistribution.

void writeDebug ( ) const

Debug write.

bool readData (Istream & *is*) `[virtual]`

ReadData function required for regIOobject read operation.

bool writeData (Ostream & *os*) const `[virtual]`

WriteData function required for regIOobject write operation.

bool operator== (const refinementTree & *rt*) const `[inline]`

bool operator!= (const refinementTree & *rt*) const `[inline]`

B.0.3.4 Friends And Related Function Documentation

Istream& operator>> (Istream & , refinementTree &) `[friend]`

Ostream& operator<< (Ostream & , const refinementTree &) `[friend]`

B.0.3.5 Member Data Documentation

List<refinementNode> children_ `[private]`

Storage for the root nodes children.

List<refinementNode*> leaves_ `[private]`

List for access to leaves.