Cite as: Darbar, Y.: Explanation of dynamicRefineFVMesh for adaptive mesh refinement with an extension for independent bulk and interface mesh refinement for two phase simulations.. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2022

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Explanation of `dynamicRefineFVMesh` for adaptive mesh refinement with an extension for independent bulk and interface mesh refinement for two phase simulations.

Developed for OpenFOAM-9

Author:
Yatin Darbar
University of Leeds

Peer reviewed by:
Stanislau Stasheuski
Saeed Salehi
Mark Wilson

January 15, 2023

Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it:

- How Adaptive Mesh Refinement (AMR) works in the `damBreakWithObstacle` tutorial and the mesh refinement settings in this tutorial.

The theory of it:

- The entries of the `dynamicMeshDict` for the `dynamicRefineFvMesh` class will be explained in detail with reference to the `damBreakWithObstacle` tutorial.

How it is implemented:

- How the AMR algorithm is called within a solver.
- THe details of the code that executes the mesh refinement processes
- How the user specified inputs in the `dynamicMeshDict` are used in the AMR updates

How to modify it:

- The steps required to extend the `dynamicRefineFvMesh` class to allow for multiple field refinement will be elucidated.
- How to use the created `dynamicDualRefineFvMesh` class to achieve independent mesh refinement in one phase of a two–phase flow and that the phase interface in the `damBreakWithObstacle` case.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to run standard tutorials like the `damBreakWithObstacle` tutorial.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric
- How to customize a solver and do top-level application programming.
- Basic understanding of C++ in the context of OpenFOAM object oriented programming
- Some understanding of the Volume of Fluid Method for simulating two phase flow.

Contents

Nomenclature

Acronyms

AMR	Adaptive Mesh Refinement
CFD	Computational Fluid Dynamics
RIJ	Reactive Inkjet Printing
VOF	Volume of Fluid

Chapter 1

Introduction

1.1 Background

At present, the OpenFOAM simulation code allows users to resolve only one evolving region using adaptive mesh refinement (AMR) algorithms. Further to this in OpenFOAM 9 multiple static regions with independent refinements levels can be utilised, however it is not possible to have two independent levels of refinement for different evolving regions in a simulation. This limits the efficiency of many multi–physics problems that require computational modelling. Achieving this is the primary work of this project.

Adaptive mesh refinement is a technique of changing the structure of a computational mesh in a localised area during a simulation. In many physical problems that require numerical modelling, a uniform computational mesh, does not result in a uniform accuracy in the obtained solution. AMR provides a framework in which regions of a simulation that need higher resolution to preserve the precision of the solution can be adapted, whereas regions that do not require as much resolution remain unchanged. Consider the case of high Reynolds number flow around a cylinder. A uniform mesh may not capture the vortex shedding in the wake of the cylinder. Rather than refining the whole mesh which will result in a large increase in the computational expense of the simulation, AMR allows the refinement of the mesh in just the wake of the cylinder to capture the vortices, but not adversely effect the intensity of the simulation.

The accuracy of solution is not the only way in which regions in a simulation can be identified for mesh refinement. In OpenFOAM, the user specifies a single scalar field present in the simulation. For example the user may request that regions above a threshold pressure should be refined. However currently in OpenFOAM there is no way to prescribe mesh refinement using on two (or more) fields within the simulation. For example there is not in-built method to refine both regions of high temperature and high pressure, or even regions of both low and high pressure. Here the first steps towards addressing this drawback will be explained.

There have been a number of previous studies that concern adapting the AMR in OpenFOAM. Many such investigations arise from reports in the CFD with OpenSource Software course. Early projects concern amalgamating AMR into solvers that did not at the time support dynamic mesh refinements. Kosters [1] showed that simulations run with the `dieselFoam` solver are highly mesh dependent, hence implemented dynamic mesh refinements into the solver, in order to better resolve the key physics present in the simulation. Similarly Nygren [2] added AMR to a moving mesh within the `sprayDyMFoam` solver. Both of these reports were published in a time when AMR was only implemented in a small selection of solvers, hence these projects mainly concerned understanding a certain solver and where to implement the AMR code rather then adapting the AMR process itself. More recently, Lindblad [3] implemented a run–time mesh refinement for the $k - \omega$ SST DES turbulence model. This extension to the OpenFOAM source code was applied to the study of flow past an aerofoil. This report gave a concise description of the `update` function used to achieve AMR in a simulation, which provides a basis for a more thorough explanation of the AMR code to be presented. One main focus of the current study is to use the AMR source code to justify the description of the parameters that the user can set to carry the dynamic mesh refinements in an OpenFOAM simulation. Finally, Eltard-Larsen [4] completed a similar task to Nygren [2] by merging two existing OpenFOAM dynamic mesh classes. Nygren merged the `dynamicMotionSolverFvMesh` and the `dynamicRefineFvMesh` classes to form the `dynamicMotionRefineFvMesh` class. This report demonstrated a solid methodology of first understanding the two dynamic mesh classes, then exemplifying in detail how to merge the two classes together. A similar process will be explored in this project in order to create a class in which AMR on two independent fields is possible. Most notably, Tobias Holzmann [5] adapted the `dynamicRefineFvMesh` class in a way such that it it possible to use two parameter sets for refining the interface and bulk of a two phase simulation in two different ways. This bespoke class was originally posted on his personal website [6] however has since been taken down, therefore the exact changes to the code are no longer available. Further to this Rettenmaier *et al* [7] contributed heavily to advancing AMR capabilities of OpenFOAM, by introducing load balanced 2D and 3D adaptive mesh refinement in OpenFOAM. The key deliverable from this work was the ability to re–decompose an adaptive mesh throughout the simulation to ensure that an even distribution of memory on the processors used in a parallel simulation. Despite this focus, the library created for this output contained amendments to the AMR implementation, which allowed users to refine the mesh simultaneously using the gradient or the curl of a vector field present in the simulation. The work presented here provides inspiration for future directions in which the developments here can be adapted.

The purpose of this document is to elucidate the AMR code in OpenFOAM in order to allow readers to follow the modifications described in Chapter 4 of this document in order to achieve independent adaptive mesh refinement on two fields in a computational fluid dynamics (CFD) simulation. In particular this adaptation will be used to achieve two distinct levels of mesh refinement in a two–phase flow simulation. The bulk of one of the phases of interest will be refined more than the other phase and in addition to this the interface between the two phases will be subject to mesh refinements. Further to this the AMR code will be adapted to support an alternative method to unrefining the mesh. This itself will also be a novel contribution to the OpenFOAM source code.

In order to achieve this extension to the existing source code, the current AMR capabilities will be illustrated using a standard OpenFOAM tutorial case. After this a detailed explanation of the AMR code will be given in order to understand how mesh refinements are carried out by an example OpenFOAM solver. This will aid in understanding how the user inputs are used by the AMR algorithm. Then, the modifications to the AMR code will be undertaken; with description of the additions, in order to extend the current AMR method to allow for refinement on two fields. Finally the tutorial case will be revisited, and the new AMR functionality will be demonstrated and explained.

1.2 Motivation

The motivation for this extension to the OpenFOAM source code stems from understanding the mixing that occurs in coalescing droplets. Chemical reactions in coalescing droplets are used in many emerging technologies to create new materials in–situ. This process is exploited heavily in the Reactive Inkjet Printing (RIJ) industry. Experimental studies are hindered by the fact that the droplets used in RIJ printing are too small to be able to visualise and understand the internal mixing dynamics [8, 9, 10]. This motivates the use of CFD in order to understand the motion within coalescing droplets. With CFD the parameter regime that RIJ processes spans can be investigated in order to better understand and quantify the mixing that occurs.

In many previous studies on coalescing droplets, the focus is on the external dynamics of the free surface [11, 12, 13]. This is since the motion of the free surface is the dominant factor controlling the movement of the droplet. Hence in many numerical studies, the focus has just been on increasing the mesh resolution at the interface the droplets. Instead of using a high resolution mesh across the whole simulation domain, in order to reduce the number of mesh elements/cells in a CFD simulation AMR is used to focus on increasing the resolution of the mesh around the free surface, without having unnecessary cells in places that do not need high resolution.

OpenFOAM allows for the AMR of cells at a fluid–fluid interface, which makes it a good choice of simulation method for two–phase flow problems like droplet coalescence. However, when studying the internal mixing dynamics of coalescing droplets the bulk fluid inside the droplet is just as important to resolve as the free surface. Hence being able to resolve the bulk to a certain level of resolution like the free surface is desirable in CFD simulations. Since the free surface dynamics influence the bulk motion of the droplets, it is advantageous to refine the interface of the droplet to a high level, and the bulk fluid of the droplet to level that is greater than the background mesh in the passive outer phase in the simulation but less than or equal to the level of the interface refinement. In this way the internal dynamics of the coalescing droplets can be accurately understood, while the computational expense of the simulation is minimised as much as possible.

1.3 Document Outline

This guide is centered around understanding the current AMR methods that are responsible for refining CFD simulations, with an extension to allow for dual-field refinement. To allow the reader to understand the motivation of this project, the existing AMR capabilities and how to create a mesh refinement class that can refine in two regions the following sections have been created.

- **Chapter 2** – A brief introduction to AMR in OpenFOAM using an example tutorial to understand the current capabilities of AMR in OpenFOAM.
- **Chapter 3** – Discussion of how the dynamic mesh is created and updated in an example solver and detailed explanation of AMR code.
- **Chapter 4** – Explanation of modifications needed to achieve dual–field refinement.
- **Chapter 5** – Demonstration of new AMR methods, revisiting the example tutorial.

Chapter 2

Using dynamicRefineFvMesh for AMR

To begin, and give motivation for the modifications that will be undertaken later, we begin by using the `damBreakWithObstacle` tutorial to illustrate how the user can employ AMR in an OpenFOAM simulation.

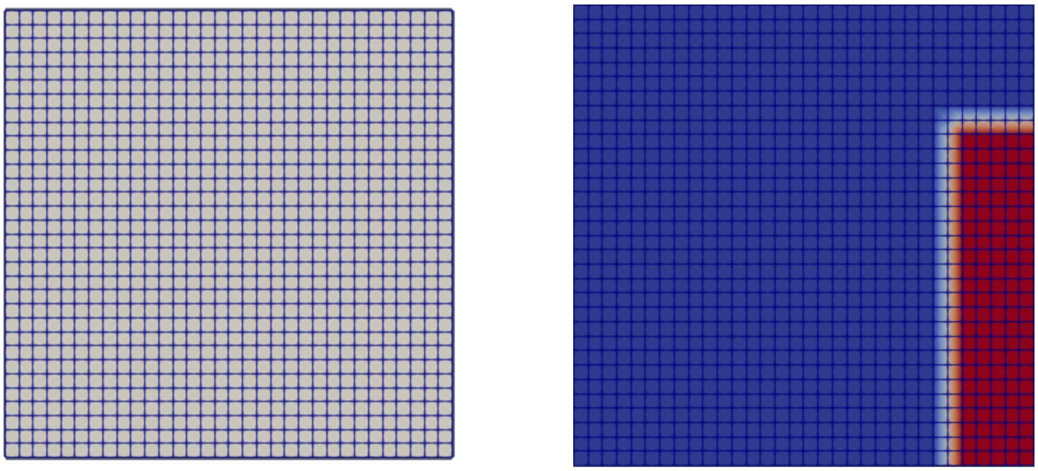
2.1 The damBreakWithObstacle Case

After sourcing OpenFOAM 9, the `damBreakWithObstacle` case can be copied to the user's working directory, in order to run the simulation, by executing the following:

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreakWithObstacle .
cd damBreakWithObstacle
./Allrun
```

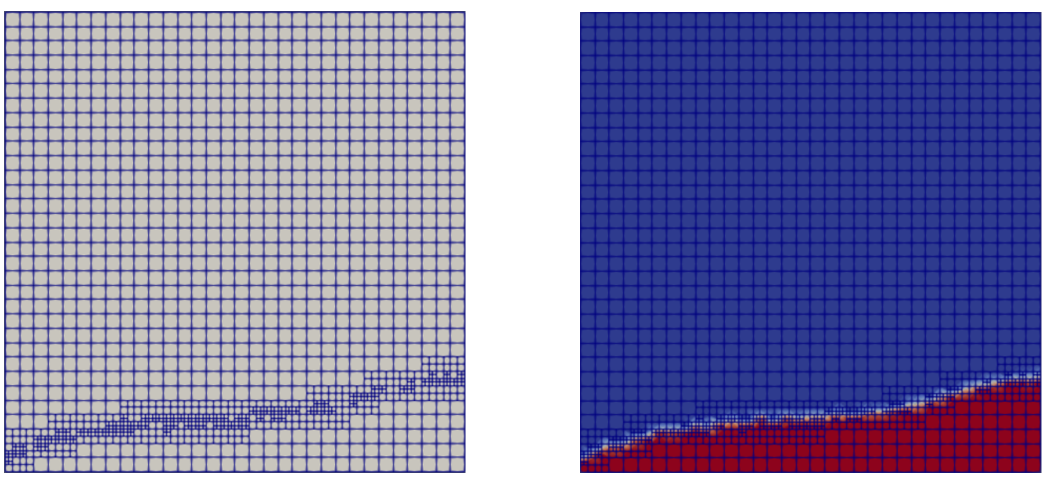
Once the simulation has completed the results can be visualised using Paraview. The AMR that takes place during the simulation can be viewed by setting the direction of the view in the camera controls to +X, the visual to be "Surface With Edges" in the Representation Toolbar and keeping the Active Variable as "Solid Colour". Figure 2.1a shows the visual that is obtained from these settings and Figure 2.1b shows the visual that is obtained by changing the active variable to `alpha.water`. Figure 2.1b shows the initial conditions of the `damBreakWithObstacle` of the simulation in which a vertical column of water is held stationary until the simulation begins and the force of gravity causes the column to collapse.

The mesh present at the start of the simulation is the uniform $32 \times 32 \times 32$ regular hexahedral mesh specified in the `system/blockMeshDict` file of the case directory, which was created when the `Allrun` script executed the `blockMesh` utility. Selecting an arbitrary write time, the changes to the mesh can be inspected. Figure 2.2a shows the mesh at 0.4 s into the simulation, and Figure 2.2b shows the mesh, with the surfaces of the mesh cells coloured by the phase fraction (`alpha.water`). At 0.4 s into the simulation the mesh is no longer a uniform grid. Instead some cells have be divided to create smaller cells. This dividing of cells is the adaptive mesh refinement. By colouring the mesh surface by the phase fraction, we see that the region coloured white/gray is the region in which the mesh has been refined. This corresponds to the region in which `alpha.water` is such that $0 <$ `alpha.water` < 1. The phase fraction `alpha.water` is used in the Volume of Fluid (VOF) method to distinguish between the phases, present in a multiphase flow. In this case the value of the phase fraction is one in all cells that contain solely water (coloured red), zero in all cells that contain solely air (coloured blue), and cells which have a value of alpha between zero and one are said to contain the air–water interface. We can conclude that throughout the simulation the mesh is being refined



(a) Uncoloured mesh (b) Mesh coloured by phase fraction

Figure 2.1: Mesh of the `damBreakWithObstacle` tutorial at time $t = 0\,\mathrm{s}$.



(a) Uncoloured mesh (b) Mesh coloured by phase fraction

Figure 2.2: Mesh of the `damBreakWithObstacle` tutorial at time $t = 0.4\,\mathrm{s}$.

at the air–water interface. The reader is encouraged to look at data from different `writeTimes` to convince themselves of this.

The mesh has been refined throughout this simulation due to the presence of the `dynamicMeshDict` file that is present in the `constant` directory of the simulation case files. This file specifies an OpenFOAM class that refines the mesh in the simulation. The class used in this simulation is called `dynamicRefineFvMesh`. This refines and unrefines the mesh, by adding or removing points, faces and cells, in the mesh. To explain how the `dynamicRefineFvMesh` class is utilised, and understand its capabilities for mesh refinement, we shall examine and explain the entries in the `dynamicMeshDict` file.

2.2 The `dynamicMeshDict` file

The `dynamicMeshDict` file for the `damBreakWithObstacle` tutorial case is provided in Appendix A.1. The reader is encouraged to refer to this while reading the description in this section. To understand this dictionary and how the settings correspond to mesh refinements seen in the `damBreakWithObstacle` simulation, each of the entries shall be explained and their impact on the mesh refinement will be linked back to the tutorial case.

2.2.1 Sub-Class declaration

The `dynamicMeshDict` begins by indicating which `dynamicFvMesh` subclass will be used in the simulation.

```
dynamicFvMesh   dynamicRefineFvMesh;
```

For completeness the subclasses of `dynamicFvMesh` are:

- `dynamicInkjetFvMesh`
- `dynamicInterpolatedFvMesh`
- `dynamicMotionSolverFvMesh`
- `dynamicRefineFvMesh`

The entries contained within the `dynamicMeshDict` depend on which subclass is selected. The details of all the sub-classes are out of the scope for this project, since we are only concerned with understanding the `dynamicRefineFvMesh` class in order to modify it.

2.2.2 `refineInterval`

The `refineInterval` parameter controls how often the mesh is refined during the simulation. Re–meshing a simulation domain is quite a computationally expensive procedure. By giving the user the ability to reduce the frequency of mesh refinements this cost can be reduced dramtically. `refineInterval` specifies the number of time–steps that should elapse before the mesh is refined/unrefined. In the `damBreakWithObstacle` tutorial, `refineInterval` is set to 1.

```
refineInterval  1;
```

Therefore every time–step the mesh is updated.

2.2.3 `field`

This is simply the variable that will be used to decided if the mesh needs refining. In all versions of OpenFOAM it must be a scalar value, therefore variables like velocity cannot be used. Instead if the user would like to refine in regions of high velocity, one component of velocity must be used, or indeed the velocity magnitude, must be calculated and stored during the simulation. In the `damBreakWithObstacle` tutorial, the phase fraction (`alpha.water`) will be used to define the criteria in which the mesh is refined.

```
field           alpha.water;
```

It will be explained in the next section how using this field, the user can refine the mesh at the air–water interface through the simulation.

2.2.4 lowerRefineInterval and upperRefineInterval

Specifying the scalar values of `lowerRefineInterval` and `upperRefineInterval` defines the criteria in which cells are refined throughout the simulation. In general a cell will be refined if the value of the cell field specified in the `field` entry is greater than the `lowerRefineLevel` and less than the `upperRefineLevel`, i.e. for the i-th cell in the computational domain, the cell is refined if

$$\texttt{lowerRefineLevel} < \texttt{field} < \texttt{upperRefineLevel}.$$

The values of `lowerRefineLevel` and `upperRefineLevel` set in the `damBreakWithObstacle` case are 0.001 and 0.999 respectively

```
lowerRefineLevel 0.001;
upperRefineLevel 0.999;
```

hence in the tutorial, cells will be refined if they satisfy the condition

$$0.001 < \texttt{alpha.water} < 0.999.$$

Recalling that the value of the phase fraction is between zero and one in all cells that contain the free surface of a two–phase flow, this confirms the qualitative conclusion that the mesh is being refined in the regions that contain the air–water interface.

2.2.5 unrefineLevel

The `unrefineLevel` input controls the mesh unrefinement. For a point in the mesh, if the value of the field selected to refine on is less than `unrefineLevel` in all the cells that surround that point i.e.

$$\texttt{field} < \texttt{unrefineLevel},$$

then the point is removed, thus unrefining the mesh, unless the point has just been added in order to refine the mesh at this time–step in the simulation. In the `damBreakWithObstacle` tutorial the `unrefineLevel` is set to 10.

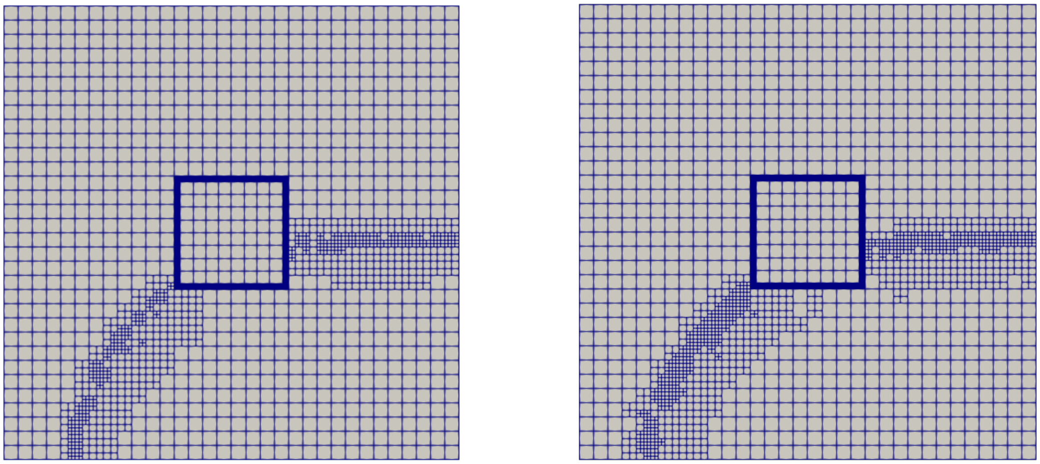
```
unrefineLevel   10;
```

Hence all points where the phase fraction is less than 10 are considered for unrefinement. Notice, from the theory of the VOF method, that the phase fraction will always be greater than or equal to zero and less than or equal to one. Therefore in this simulation all of the cells/points in the computational domain are considered for unrefinement. We shall explore in greater detail later, why the cells that fulfill the refinement criteria are not unrefined even though they satisfy the unrefinement criteria.

2.2.6 nBufferLayers

In order to avoid sharp changes in the mesh grading, the `nBufferLayers` parameter prescribes a number of layers that must "bridge" the gap between refined and unrefined regions of the mesh. The larger the value of `nBufferLayers` the larger this intermediate layer is. In the `damBreakWithTutorial` case the `nBufferLayers` parameter is set to 1.

```
nBufferLayers   1;
```



(a) `nBufferLayers` = 1 (b) `nBufferLayers` = 4

Figure 2.3: Comparison of the cells around the interface in the `damBreakWithObstacle` tutorial at 0.2 s showing the effect of increasing the `nBufferLayers` parameter.

This produces a modest but adequate layer of cells around the interface. This can be more easily seen by using the +Z camera view in Paraview. Indeed if the `nBufferLayers` parameter is increased, then the zone of cells around the interface extends. Figure 2.3 shows a comparison of the mesh at 0.2 s into the simulation between the `damBreakWithObstacle` tutorial with the `nBufferLayers` parameter set to 1 and 4.

2.2.7 maxRefinement

Examining Figure 2.2a again, it is possible to see that some of the cells from the $32 \times 32 \times 32$ mesh have been split, and split again. Before the simulation is executed a cell in the computational mesh has a `cellLevel` of zero. When a cell is refined, the value of the `cellLevel` for the resulting cells that are created is increased by one. The `maxRefinement` entry sets the maximum number of times an initial cell in the mesh before the simulation can be refined by defining the largest `cellLevel` any future mesh cell may have. In the `damBreakWithObstacle` tutorial, `maxRefinement` is set to 2

```
maxRefinement   2;
```

Therefore the initial cells in the computational mesh will be refined a maximum of twice in the simulation. This explains the observation in Figure 2.2a that some cells look like they have been split twice over.

2.2.8 maxCells

The core concept of AMR is reducing computational expense by reducing the overall number of mesh cells in the simulation. However splitting one cell results in seven new cells in the computational domain, therefore in some cases the number of cells in the simulation can grow exponentially. The `maxCells` parameter limits the total number of cells in the mesh, to ensure that the expense of the simulation is not compromised by a blow up in the number of cells. Once the total number of cells in the computational domain reaches `maxCells` refinement of the mesh stops and only unrefinement can occur until the number of cells in the computational domain is less than `maxCells`. In the `damBreakTutorial` the `maxCell` parameter is set to 200000

```
maxCells        200000;
```

There is no rubric on setting this parameter since it is highly case dependent.

2.2.9 correctFluxes

The `correctFluxes` entry in the `dynamicMeshDict` is a dictionary that contains a list of the fluxes on the cell faces in the simulation and corresponding velocity field. Fluxes on faces that change within the mesh get recalculated by interpolating the velocity field. For fluxes that do not need to be re–interpolated the `none` keyword can be used. This feature is mostly used in the mesh refinement classes that alter the mesh topology. Listing 2.1 shows an example of how the fluxes are re-calculated in the `dynamicRefineFvMesh::refine` function. This is done by taking the scalar product of the cell face normals (`Sf()`) and the interpolated value of the field at the faces of the cell (`fvc::interpolate()`).

```
    const surfaceScalarField phiU
    (
        fvc::interpolate
        (
            lookupObject<volVectorField>(UName)
        )
        & Sf()
    );
```

Listing 2.1: Example section of the code used to re–interpolate the fluxes on mesh faces

In the case of the `damBreakWithObstacle` simulation, since the initial mesh is a regular hexahedral mesh, the refinements do not cause the orientation of the faces in the mesh to be altered. Therefore no re–interpolation is required. The `correctFluxes` table is given by

```
correctFluxes
(
    (phi none)
    (nHatf none)
    (rhoPhi none)
    (alphaPhi0.water none)
    (ghf none)
);
```

2.2.10 dumpLevel

The last entry in the `dynamicRefineDict` is a boolean input called, `dumpLevel`. This input gives the user the optional to write out the `cellLevel` for each cell in the simulation. This allows the user to visualise the `cellLevel` in Paraview alongside the other standard simulation data such velocity, pressure etc. The code that executes this is found in the `dynamicRefineFvMesh::writeObject` function

```
     if (dumpLevel_)
    {
        volScalarField scalarCellLevel
        (
            IOobject
            (
                "cellLevel",
                time().timeName(),
                *this,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE,
                false
            ),
            *this,
            dimensionedScalar(dimless, 0)
        );

        const labelList& cellLevel = meshCutter_.cellLevel();

        forAll(cellLevel, celli)
```

```
        {
            scalarCellLevel[celli] = cellLevel[celli];
        }

        writeOk = writeOk && scalarCellLevel.write();
    }
```

An `IOobject` called `cellLevel` that will be written out during the simulation is created and the `cellLevel()` member function of the `hexRef8` classes is used to access the `cellLevel` of all cells in the mesh. The `cellLevel` obtained from the mesh is then stored in the `IOobject cellLevel` which is written out in the simulation. This functionality is convenient since it allows the user to assess whether AMR is suited for the simulation problem. If the regions of high `cellLevel` are quite static, then a graded mesh or an over–set mesh, may be more appropriate for the simulation, rather than a dynamically evolving mesh.

Now that we have investigated an example of using AMR in a multiphase simulation and understood how to control the mesh refinements on the user level, we see that refinement using only one field is possible. Therefore the next step is to understand how the source code of the solver invokes the `dynamicFvMesh` class and updates the mesh during the simulation. After which we will be in a position to make amendments to the source code.

Chapter 3

Dynamic meshing code

Before examining the `dynamicRefineFvMesh` class, in order to understand how AMR is achieved in OpenFOAM, we shall begin with a top down approach to identify the lines of code that initialise the dynamic mesh and cause it to be updated. In this way we can restrict our attention to the core pieces of code that need to be examined.

When a simulation is carried out using OpenFOAM, we execute only the solver to begin the simulation. Therefore the solver must carry out the AMR routine. Examining the solver source code will allow us to determine how the mesh refinements are accomplished in an OpenFOAM simulation. Since in the `damBreakWithObstacle` tutorial the `interFoam` solver is used, we shall use this solver as an example to examine how dynamic mesh refinements are hard–coded into the solver source code. Though we examine the `interFoam` solver specifically in this report, we shall see that the AMR routine is called in a general way, which is very similar across many of the OpenFOAM solvers that that boast AMR capabilities.

3.1 The `interFoam` source code

The code that defines the `interFoam` solver is found in the `interFoam.C` file. The full `interFoam.C` file can be found in Appendix B.1. To understand how the dynamic mesh is created in the simulation, it is first useful to examine the files that are included within the beginning of the source code. Listing 3.1 shows the main files included within the `interFoam.C` file.

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "CMULES.H"
#include "EulerDdtScheme.H"
#include "localEulerDdtScheme.H"
#include "CrankNicolsonDdtScheme.H"
#include "subCycle.H"
#include "immiscibleIncompressibleTwoPhaseMixture.H"
#include "noPhaseChange.H"
#include "kinematicMomentumTransportModel.H"
#include "pimpleControl.H"
#include "pressureReference.H"
#include "fvModels.H"
#include "fvConstraints.H"
#include "CorrectPhi.H"
#include "fvcSmooth.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

```

```
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"
    #include "createFields.H"
    #include "createFieldRefs.H"
    #include "createAlphaFluxes.H"
    #include "initCorrectPhi.H"
    #include "createUfIfPresent.H"
```

Listing 3.1: The included files in the `interFoam` solver

From these files we can begin to identify and investigate the generation of the dynamic mesh.

3.1.1 `dynamicFvMesh.H`

The first file that concerns the creation of the dynamic mesh, is the `dynamicFvMesh.H` header file. This file is included in order to define the base class `dynamicFvMesh`, which all the dynamic mesh classes listed in Section 2.2.1 inherit from. The `dynamicFvMesh` itself inherits from the `fvMesh` class, which is the class used for standard static mesh simulations. `dynamicFvMesh` extends upon the capabilities of the `fvMesh` class by providing the features that allow for a changing mesh during the run time of a simulation. An example of such features is being able to read and return the dictionary that defines the changes to the dynamic mesh, and also an update function that will execute changes to the dynamic mesh for mesh motion and topological mesh changes.

3.1.2 `createDynamicFvMesh.H`

This inclusion allows the addition of a small section of code that calls the `dynamicFvMesh::New` function, which is defined in `dynamicFvMeshNew.C`. The section of code is presented in Listing 3.2 The `dynamicFvMesh::New` function creates a dynamic mesh object by reading the `dynamicFvMesh` sub–class that is specified in the `dynamicMeshDict` file in the case directory and creating a mesh of that type.

```
Info<< "Create mesh for time = "
     << runTime.timeName() << nl << endl;

 autoPtr<dynamicFvMesh> meshPtr
 (
     dynamicFvMesh::New
     (
         IOobject
         (
             dynamicFvMesh::defaultRegion,
             runTime.timeName(),
             runTime,
             IOobject::MUST_READ
         )
     )
 );

 dynamicFvMesh& mesh = meshPtr();
```

Listing 3.2: `createDynamicFvMesh.H` file

For example in the `damBreakWithObstacle` case because the class `dynamicRefineFvMesh` was declared at the beginning of the `dynamicMeshDict`, a `dynamicRefineFvMesh` object was created. After the `dynamicfvMesh::New` function is called, a reference called `mesh` is created. This reference is to the `dynamicFvMesh` type that was just created by the `dynamicfvMesh::New` function. With this in mind when the object `mesh` is used within the code from now on, we understand that it is a reference to an object from a subclass of `dynamicFvMesh`

3.1.3 createDyMControls.H

Again, this included file is to insert a small section of code into the solver. Namely,

```
#include "createControl.H"
#include "createTimeControls.H"

bool correctPhi
(
    pimple.dict().lookupOrDefault("correctPhi", mesh.dynamic())
);

bool checkMeshCourantNo
(
    pimple.dict().lookupOrDefault("checkMeshCourantNo", false)
);

bool moveMeshOuterCorrectors
(
    pimple.dict().lookupOrDefault("moveMeshOuterCorrectors", false)
);
```

Listing 3.3: createDyMControls.H file

In this case the section of code concerns reading some of the settings from the PIMPLE dictionary from the case directory in the system/fvSolution file. The code creates three boolean objects:

- correctPhi
- checkMeshCourantNo
- moveMeshOuterCorrectors

The value of these boolean objects is determined by using the lookupOrDefault function. If any of these keywords are found in the PIMPLE dictionary, then the value from the dictionary is assigned to the value here, if they are not found in the dictionary, then the values are assigned a default value specified in this section of code. The details of these controls are beyond the scope of this project, so more detail will not be provided.

3.1.4 Time Loop

After the inclusion of the files presented in Listing 3.1, the dynamic mesh has been selected, constructed and made accessible through the mesh reference object. After which the time loop begins (line 78 of interFoam.C) and within the time loop first the dynamic mesh controls that were created in the createDyMControls.H file are read for use within the solution procedure.

```
78      Info<< "\nStarting time loop\n" << endl;
79
80      while (pimple.run(runTime))
81      {
82          #include "readDyMControls.H"
83
84          if (LTS)
85          {
86              #include "setRDeltaT.H"
87          }
88          else
89          {
90              #include "CourantNo.H"
91              #include "alphaCourantNo.H"
92              #include "setDeltaT.H"
93          }
94
95          runTime++;
96
```

```
97          Info<< "Time = " << runTime.timeName() << nl << endl;
98
99          // --- Pressure-velocity PIMPLE corrector loop
100         while (pimple.loop())
101         {
102             if (pimple.firstPimpleIter() || moveMeshOuterCorrectors)
103             {
104                 // Store divU from the previous mesh so that it can be mapped
105                 // and used in correctPhi to ensure the corrected phi has the
106                 // same divergence
107                 tmp<volScalarField> divU;
108
109                 if
110                 (
111                     correctPhi
112                  && !isType<twoPhaseChangeModels::noPhaseChange>(phaseChange)
113                 )
114                 {
115                     // Construct and register divU for mapping
116                     divU = new volScalarField
117                     (
118                         "divU0",
119                         fvc::div(fvc::absolute(phi, U))
120                     );
121                 }
122
123                 fvModels.preUpdateMesh();
124
125                 mesh.update();
```

Listing 3.4: The beginning of the `interFoam` time–loop

The reason the mesh controls are re–read at the start of each time–step is to allow the user the ability to change the dynamic mesh controls in the PIMPLE dictionary during the run time of the simulation. An advantage of OpenFOAM is the flexibility in having many run-time modifiable settings. Next, the time step is set, the Courant number is calculated and this information along with the current simulation time is printed to the output stream. After this the first call to the `mesh` reference is on line 125 of the `interFoam.C` code. On this line the `update()` function is executed, which is a member function of the `mesh` object. Since this is a pointer to the `dynamicFvMesh` subclass that was created in `createDynamicFvMesh.H` file, it is seen that the code `mesh.update()` is really a call to the `dynamicFvMesh::update()` function.

In the case of the `damBreakWithObstacle` tutorial, since a `dynamicRefineFvMesh` class is declared as the type for the dynamic mesh, then the `mesh` object created in the `createDynamicFvMesh.H` file is a reference to a `dynamicRefineFvMesh` object. Therefore, when the `mesh.update()` function is called, in actuality the `dynamicRefineFvMesh::update()` function is called. Since this project concerns making modifications to the `dynamicRefineFvMesh` class, we shall restrict attention to the `dynamicRefineFvMesh::update()` function, in order to understand what this function does and how it accomplishes AMR within `dynamicRefineFvMesh` class, before going on to make modifications to this class.

3.2 `dynamicRefineFvMesh::update()`

As will be elucidated, the `dynamicRefineFvMesh::update()` function is responsible for carrying out the dynamic mesh refinements when the `dynamicRefineFvMesh` class is used. For brevity, henceforth the function will be referred to as the `update()` function, since this report concerns the `dynamicRefineFvMesh` class. A flow chart outlining the key steps in the function is provided in Figure 3.1. A detailed description of theses key steps will be given in order to understand how the mesh refinement is carried out, when using the `dynamicRefineFvMesh` class and corresponding `dynamicMeshDict`.

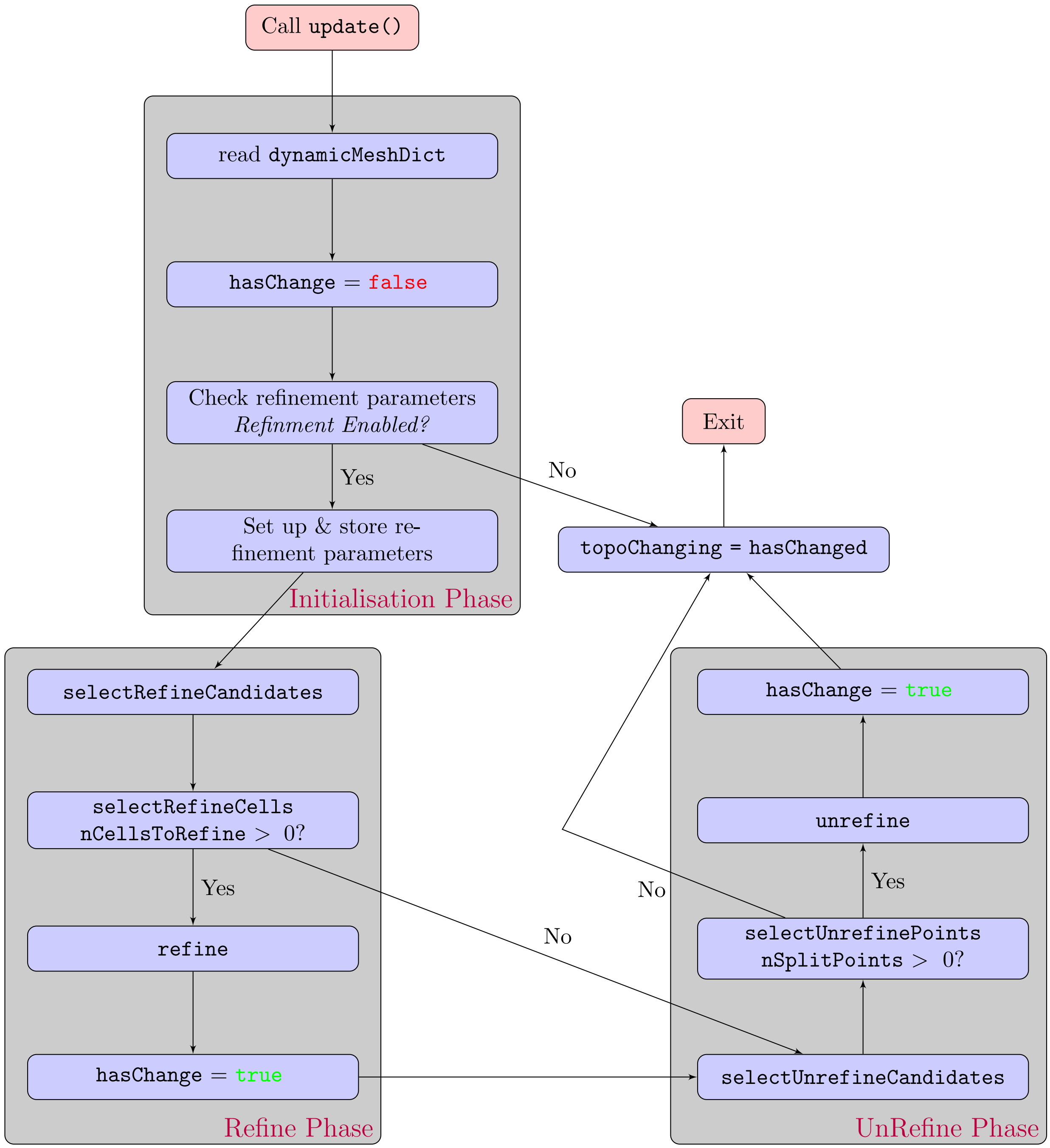


Figure 3.1: A flow diagram, outlining the key process and steps in the `dynamicRefineFvMesh::update` function.

3.2.1 Initialisation Phase

To begin the `update()` function reads in the `dynamicMeshDict` from the case directory and stores it locally as a `dictionary` called `refineDict`.

```
    const dictionary refineDict
    (
        dynamicMeshDict().optionalSubDict(typeName + "Coeffs")
    );
```

Notice that the `dynamicMeshDict` is re–read every time–step (since the `update()` function is called each time–step) in order to allow for run–time modifications to the AMR strategy throughout the simulation.

In order to indicate when adaptations to the mesh have been made the function uses a `boolean` variable `hasChanged`. This is initially set to `false` since during the initialisation phase of the function no changes to the mesh have been undertaken.

```
    bool hasChanged = false;
```

We shall see in later phases of the `update()` function the value of `hasChanged` is reassigned when alterations to the mesh have been completed.

The next stage of the initialisation phase is to check the refinement parameters. This is done by looking up the entries in the `refineDict` (recall this is a local copy of `dynamicMeshDict` at a given time–step). First a `label` for the `refineInterval` is created and assigned the value of the `refineInterval` in the `refineDict`.

```
    label refineInterval = refineDict.lookup<label>("refineInterval");
```

The function now proceeds to determine whether mesh refinement is enabled.

```
    if (refineInterval == 0)
    {
        topoChanging(hasChanged);

        return false;
    }
    else if (refineInterval < 0)
    {
        FatalErrorInFunction
            << "Illegal refineInterval " << refineInterval << nl
            << "The refineInterval setting in the dynamicMeshDict should"
            << " be >= 1." << nl
            << exit(FatalError);
    }
```

If the refinement variable `refineInterval` is set to zero then mesh refinements are disabled and the function ceases operation. This is useful in situations such as finding a steady state solution before enabling refinements to the mesh. If the `refineInterval` is less than zero, a warning will be passed to the output stream since this is an invalid value for `refineInterval` and the solver will also terminate execution. In the case that `refineInterval` is defined appropriately (`refineInterval` is greater than zero), then the function moves on to storing a local version of the `maxCell` parameter.

```
        label maxCells = refineDict.lookup<label>("maxCells");
```

Again this parameter is checked in order to make sure that it is properly defined since a non-positive value of `maxCells` does not make physical sense.

```
        if (maxCells <= 0)
        {
            FatalErrorInFunction
                << "Illegal maximum number of cells " << maxCells << nl
                << "The maxCells setting in the dynamicMeshDict should"
```

```
1367                << " be > 0." << nl
1368                << exit(FatalError);
1369        }
```

The final part of the initialisation phase is to create and store a local value of the the `nBufferLayers` parameter

```
1371        const label nBufferLayers =
1372            refineDict.lookup<label>("nBufferLayers");
```

3.2.2 Refinement Phase

The refinement phase of the `update()` function begins with the identification of which cells in the computational mesh should be refined based on the criteria set out in the `refineDict` and the assignment of the `maxRefinement` parameter. These two tasks are both carried out by the `dynamicRefineFvMesh::selectRefineCandidates` function. The `selectRefineCandidates` functions are defined on lines 684-804 of the `dynamicRefineFvMesh.C` file (See Appendix B.2). The `maxRefinement` parameter is assigned and checked in a similar way to the `refineInterval` and `maxCells` parameters in the initialisation phase.

```
766     const label maxRefinement = refineDict.lookup<label>("maxRefinement");
```

It too cannot have a non-positive value, hence an error warning is output in the cases when the user specifies an invalid value of the `maxRefinement` parameter.

```
768     if (maxRefinement <= 0)
769     {
770         FatalErrorInFunction
771             << "Illegal maximum refinement level " << maxRefinement << nl
772             << "The maxCells setting in the dynamicMeshDict should"
773             << " be > 0." << nl
774             << exit(FatalError);
775     }
```

As well as returning the `maxRefinement` value, this function creates a lists of cells that are candidates for refinement. The `selectRefineCandidates` identifies cells as candidates for refinement by using the local `error` function (defined on lines 635–655 of the `dynamicRefineFvMesh.C` file). The error of each cell in the computational mesh is calculated by the function

$$\text{error}_i = \min\{\texttt{field}_i - \texttt{lowerRefineLevel}, \texttt{upperRefineLevel} - \texttt{field}_i\}, \tag{3.1}$$

where the subscript i denotes the i–th cell in the computational mesh. The error function gives a way to quantify the closest distance to either of the `upperRefineLevel` and `lowerRefineLevel` limits. This information is not used elsewhere in the update function, however it lays the foundation for prioritising cells with higher error to be the ones refined first. Since these cells are furthest from both `upperRefineLevel` and `lowerRefineLevel`. As we shall see later this will be beneficial in cases in which all the candidates for refinement cannot be refined due to the maximum allowable number of cells in the mesh being exceed. The `selectRefineCandidates` function identifies cells that should be refined, by measuring the cells error and its `cellLevel`. If the error of the cell is great than zero, and the `cellLevel` is less than the `maxRefinement` parameter then the cell is identified as a candidate to be refined.

```
695     const scalarField cellError
696     (
697         error(vFld, lowerRefineLevel, upperRefineLevel)
698     );
699
700     const labelList& cellLevel = meshCutter_.cellLevel();
701
702     // Mark cells that are candidates for refinement.
703     forAll(cellError, celli)
```

```
704     {
705         if
706         (
707             cellLevel[celli] < maxRefinement
708          && cellError[celli] > 0
709         )
710         {
711             candidateCells.set(celli, 1);
712         }
713     }
```

From the definition of the error function in equation 3.1, we can we see that the cell error is non–negative whenever

$$\texttt{lowerRefineLevel} < \texttt{field} < \texttt{upperRefineLevel}. \tag{3.2}$$

which confirms the statement in Section 2.2.4.

Before beginning the actual refinement procedure and turning the list of candidate cells for refinement into a definitive list of cells that will be refined, a trivial check that the number of cells in the computational mesh domain is less than `maxCells` is completed.

```
1404        if (globalData().nTotalCells() < maxCells)
```

If this is not the case then the refinement process is skipped and the function proceeds to the unrefinement sub–routine. If the number of cells in the simulation is less than `maxCells`, then the refinement takes places. In order to avoid undergoing a mesh refinement that would result in creating too many cells, The number of cells that can be refined is calculated, assuming that each refined cell creates seven new cells. This is done by using the `selectRefineCells` function (defined on lines 807–891 of `dynamicRefineFvMesh.C`)

```
815     label nTotToRefine = (maxCells - globalData().nTotalCells()) / 7;
```

After which the number of cells that have been marked for refinement is checked against this estimation

```
831     if (nCandidates < nTotToRefine)
```

There are two situations that can occur. The first being that the number of cells after refinement will not exceed `maxCells`. In this case refinement will occur to all marked cells. The second case occurs when the number of cells after refinement is more than `maxCells`. In this case, cells are refined until the number of cells in the mesh becomes larger than `maxCell`. The order in which cells are chosen in this case is just in the order they are listed in the candidates array. There is scope here to use the data gathered from the `error` function in order to rank the cells that should be refined first. This final check gives a list `nCellsToRefine` which contains all the cells that will be refined this time–step.

If `nCellsToRefine` is greater than zero, the refinement procedure begins by calling the `refine` function. Briefly this function creates the new mesh by splitting the cells that need to be refined by introducing a point in the middle of the cell. The cell fields are then mapped and the flux is approximated on the newly created faces. Notice that the flux correction will only occur if specified in the `correctFluxes` dictionary in the `dynamicMeshDict` file.

At the end of the refinement procedure, the `boolean` `hasChanged` is set to `true`, since in the case that `nCellsToRefine` is greater than zero, the mesh has been updated. The function now moves onto the unrefinement sub–routine.

```
1463                hasChanged = true;
```

3.2.3 Unrefinement Phase

The phase starts by selecting the points that should be unrefined. This is done by calling the `selectUnrefineCandidates` function (defined on lines 894 –983 of `dynamicRefineFvMesh.C`). It is noted here that when talking about unrefinement of the mesh, a computational cell itself cannot be unrefined, but rather a common point/corner of a set of cells can be removed to create a larger cell. Hence when selecting candidates to unrefine the mesh, we consider points in the computational mesh, rather than cells.

The `selectUnrefineCandidates` function works by considering the cells around a point in the mesh.

```
901     forAll(pointCells(), pointi)
902     {
903         const labelList& pCells = pointCells()[pointi];
904 
905         scalar maxVal = -great;
906         forAll(pCells, i)
907         {
908             maxVal = max(maxVal, vFld[pCells[i]]);
909         }
910 
911         unrefineCandidates[pointi] =
912             unrefineCandidates[pointi] && maxVal < unrefineLevel;
913     }
```

It finds the largest value of the field that has been selected to control the refinement of the mesh (`field`) in the cells surrounding a point and stores that as the variable `maxVal`. Then if

$$\texttt{maxVal} < \texttt{unRefineLevel}, \tag{3.3}$$

the point is made a candidate for removal, thereby giving a way to unrefine the mesh by removing points.

Again, not all the candidates for unrefinement are passed on to actually be unrefined, in this case the `selectUnrefinePoints` function is used to ensure that points that have been identified as candidates to be removed are not part of protected cells, or the intermediate layer that is created to have guarantee a smooth mesh level transition between refined and unrefined regions. The `selectUnrefinePoints` function also takes the `refineCells` array as an input, to ensure that any cells that have been refined on this iteration of the mesh refinement algorithm are not immediately unrefined in the same time–step. This is why in the `damBreakWithObstacle` tutorial despite setting all cells to be unrefined (by setting `unRefineLevel` to 10), those cells that have been refined are not immediately unrefined in the same time–step. Once the `selectUnrefinePoints` function has approved the points that can be removed from the mesh, the finalised list of points that are to be removed are stored in the `nSplitPoints` variable. Another trivial check that the number of `nSplitPoints` is greater than zero is completed before executing the unrefinement procedure.

```
1511            if (nSplitPoints > 0)
```

The `unrefine` function executes the removal of the `nSplitPoints` in the mesh. As before, the fields are also mapped and the fluxes are recreated approximately on the new faces. Also as before, the fluxes will only be recreated if they are listed under `correctFluxes` in the `dynamicMeshDict`. The full details of the `unrefine` function are out of the scope for this project.

The last stage of the unrefinement procedure is to change the boolean `hasChanged` to true. This could already have a `true` value the mesh has been refined during the refinement phase, but since the `update` function allows for refinement without unrefinement and vice versa, the `hasChanged` value must also be changed here also.

```
1516                hasChanged = true;
```

After the unrefinement procedure, the update function then passes the value of `hasChanged` to the `topoChanging` boolean. The `topoChanging` variable is returned by the call to `mesh.update()` and the value is used later on in the `interFoam.C` code.

This concludes the description of the `dynamicRefineFvMesh::update()` function. It is now possible to adapt the `dynamicRefineFvMesh` class in order to add the capabilities to refine on two fields within the CFD simulation. The instructions and details of the adaptations necessary are described in the next chapter.

Chapter 4

Creating dynamicDualRefineFvMesh

4.1 Introduction

In order to establish the ability to refine on two fields in a CFD simulation using OpenFOAM a new `dynamicFvMesh` subclass needs to be implemented. This will primarily be achieved by adapting the the source code of the `dynamicRefineFvMesh` class and building on the mesh refinement algorithms and processes described in Chapter 3. The new `dynamicFvMesh` subclass will be named `dynamicDualRefineFvMesh`, since the core principle of the new class is to add the functionality to refine on two fields in a simulation. The report shall focus on using the `dynamicDualRefineFvMesh` class to refine the mesh of a separated two-phase flow, but reader is encouraged to apply the `dynamicDualRefineFvMesh` class to other investigations. The finished class is provided in the accompanying files, but the reader is encouraged to understand and carry out the changes to the source code.

We shall begin by copying the `dynamicRefineFvMesh` source code, compiling into our own library, then checking that this library is accessible by running the `damBreakWithObstacle` tutorial again. After which the alterations to the code will be undertaken and the `dynamicDualRefineFvMesh` will be created and compiled.

4.2 Creating dynamicDualRefineFvMesh

In order to create a user modifiable version of `dynamicRefineFvMesh` which will be the basis of `dynamicDualRefineFvMesh`, we shall make a copy of the `dynamicRefineFvMesh` directory (located in `$FOAM_SRC/dynamicFvMesh`) in the user directory. To do this we can execute the following commands (after sourcing the OpenFOAM).

```
cd $WM_PROJECT_USER_DIR
mkdir src/dynamicFvMesh/dynamicDualRefineFvMesh
cd src/dynamicFvMesh/dynamicDualRefineFvMesh
cp -r $FOAM_SRC/dynamicFvMesh/dynamicRefineFvMesh/dynamicRefineFvMesh* .
```

We shall begin by renaming the class files to match the new class that will be implemented as part of this report, as well as renaming all the occurrences of the class names in the header and main files. Now we must rename the class files and their occurrences.

```
mv dynamicRefineFvMesh.H dynamicDualRefineFvMesh.H
mv dynamicRefineFvMesh.C dynamicDualRefineFvMesh.C
sed -i 's/dynamicRefineFvMesh/dynamicDualRefineFvMesh/g' dynamicDualRefineFvMesh.*
```

In order to compile this class, we must create a `Make` directory with the corresponding `files` and `options` files. The simplest way to create this is to copy the `Make` folder that compiles the `dynamicRefineFvMesh` class located in the `$FOAM_SRC/dynamicFvMesh` directory

```
cp -r $FOAM_SRC/dynamicFvMesh/Make .
```

Notice that the copied `Make/files` file, is used to compile `dynamicFvMesh` and all of its sub–classes. Hence, in order to compile solely the user created `dynamicDualRefineFvMesh` class, the `Make/files` file must be changed to

Make/files

```
dynamicDualRefineFvMesh.C

LIB = $(FOAM_USER_LIBBIN)/libdynamicDualRefineFvMesh
```

Here we also note that the library is saved in the `$FOAM_USER_LIBBIN` since users do not have the ability to compile libraries in the `$FOAM_LIBBIN`. Since we have moved the `dynamicDualRefineFvMesh` library away from the base class `dynamicFvMesh`, we must ensure that the compiled base class can be accessed. Therefore an additional set of lines needs to be added to the `Make/options` file:

Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/triSurface/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude

LIB_LIBS = \
    -ltriSurface \
    -lmeshTools \
    -ldynamicMesh \
    -lfiniteVolume \
    -ldynamicFvMesh
```

Finally, it is possible to ensure that any users are aware that a non-standard library is being used in the simulation. The `dynamicDualRefineMesh::update()` function can be amended to print out a statement to the output stream to alert the user that the `dynamicDualRefineFvMesh` is being used.

```
bool Foam::dynamicDualRefineFvMesh::update()
{
    Info<< "Using dynamicDualRefineFvMesh" << endl;
```

We can now compile the code and check that this class is accessible by the OpenFOAM solver. Recall to compile the class we can use

```
wclean
wmake
```

4.2.1 Testing

Since no modifications to the bespoke class have been made apart from changing the name of the class, the `dynamicDualRefineFvMesh` class should run and produce just as the `dynamicRefineFvMesh` class did in Chapter 2. In order to test that the renaming and recompiling procedure has resulted in a new class that is accessible by the OpenFOAM solvers, the `damBreakWithObstacle` tutorial will be executed. In order to use the newly created class, we can copy a new version of the `damBreakWithObstacle` tutorial in the users run directory and call it `testDamBreak`.

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreakWithObstacle ./testDamBreak
cd testDamBreak
```

To use the newly created `dynamicDualRefineFvMesh` the `dynamicFvMesh` type specified in the `dynamicMeshDict` must be changed from `dynamicRefineFvMesh` to `dynamicDualRefineFvMesh`.

```
sed -i 's/dynamicRefineFvMesh/dynamicDualRefineFvMesh/g' constant/dynamicMeshDict
```

And the `dynamicDualRefineFvMesh` library must be added to the `system/controlDict` file

```
echo 'libs ("libdynamicDualRefineFvMesh.so");' >> system/controlDict
```

Since the objective of this part of the tutorial is only to ensure that the `dynamicDualRefineFvMesh` is accessed by the `interFoam` solver, we shall forego creating the obstacle that is part of the original `damBreakWithObstacle` simulation. To test the access of the class we run

```
blockMesh
setFields
interFoam
```

From the output stream we can see that the "Using dynamicDualRefineFvMesh" message is printed, and hence the `interFoam` solver is using the user compiled `dynamicDualRefineFvMesh` class in order to conduct the mesh refinement in the simulation.

4.3 Adding Dual–Field Refinement

4.3.1 Methodology

Despite the motivation for this project stemming from the desire to have a refined interface and bulk phase in multiphase simulation, the adaptations to the source code can be applied to a myriad of situations. Hence what will be implemented is a class that can handle mesh refinements on in two regions, defined by the same or independent scalar fields present in the CFD simulation. The method of generating dual–field refinement, will be to create two distinct parts to the `update` function, one section will refine and unrefine one field, and the latter part will focus on refining and unrefining the second field. For ease of explanation, the changes to the variables in the source code will be denoted with a suffix `1` or `2`, depending on which of the fields the code concerns. The changes made in this report are not the most efficient implementation to produce the required result, however they are seen as first proof of concept implementation to demonstrate the capabilities of OpenFOAM, in part to understand and prove that minimal changes to the code are needed, just a reshaping of the mesh refinement algorithms. Further to this the current implementation will not be capable of dealing with the `refinementRegions` that one can define in OpenFOAM-9 for static regions of mesh refinement. Therefore the following adaptations will remove and not consider some of the differences to the code that this entails.

4.3.2 Initialisation Phase

As discussed in Section 3.2.1, the `update` function which conducts the mesh refinement undergoes an initialisation phase, in which the parameters specified in the `constant/dynamicMeshDict` are read, stored and checked to ensure they are correctly defined. The first step towards implementing two field refinement is to amend the initialisation phase of the `update` function in order to ensure that the parameters for both fields that will be refined on during the simulation are well–defined. In this implementation parameters for each field that will be refined on must be created. For reference the original initialisation phase of `dynamicRefineFvMesh` is given in Listing 4.1.

```
    const dictionary refineDict
    (
        dynamicMeshDict().optionalSubDict(typeName + "Coeffs")
    );

    label refineInterval = refineDict.lookup<label>("refineInterval");
```

```

    bool hasChanged = false;

    if (refineInterval == 0)
    {
        topoChanging(hasChanged);

        return false;
    }
    else if (refineInterval < 0)
    {
        FatalErrorInFunction
            << "Illegal refineInterval " << refineInterval << nl
            << "The refineInterval setting in the dynamicMeshDict should"
            << " be >= 1." << nl
            << exit(FatalError);
    }

    // Note: cannot refine at time 0 since no V0 present since mesh not
    //       moved yet.

    if (time().timeIndex() > 0 && time().timeIndex() % refineInterval == 0)
    {
        label maxCells = refineDict.lookup<label>("maxCells");

        if (maxCells <= 0)
        {
            FatalErrorInFunction
                << "Illegal maximum number of cells " << maxCells << nl
                << "The maxCells setting in the dynamicMeshDict should"
                << " be > 0." << nl
                << exit(FatalError);
        }

        const label nBufferLayers =
            refineDict.lookup<label>("nBufferLayers");
```

Listing 4.1: Original Initialisation phase

Whereas, in the `dynamicDualRefineFvMesh.C` file, the initialisation of the `update` function is given in Listing 4.2.

```
    Info<< "Using dynamicDualRefineFvMesh" << endl;
    // Re-read dictionary. Chosen since usually -small so trivial amount
    // of time compared to actual refinement. Also very useful to be able
    // to modify on-the-fly.
    const dictionary refineDict
    (
        dynamicMeshDict().optionalSubDict(typeName + "Coeffs")
    );

    label refineInterval1 = refineDict.lookup<label>("refineInterval1");
    label refineInterval2 = refineDict.lookup<label>("refineInterval2");

    bool hasChanged = false;

    if (refineInterval1 == 0 && refineInterval2 == 0)
    {
        topoChanging(hasChanged);

        return false;
    }
    else if (refineInterval1 < 0  || refineInterval2 < 0)
    {
        FatalErrorInFunction
            << "Illegal refineInterval " << refineInterval1
            << " | " << refineInterval2 << nl
            << "The refineInterval setting in the dynamicMeshDict should"
```

```
            << " be >= 1." << nl
            << exit(FatalError);
    }
    // Note: cannot refine at time 0 since no V0 present since mesh not
    //       moved yet.

    if (time().timeIndex() > 0
    && time().timeIndex() % refineInterval1 == 0
    && time().timeIndex() % refineInterval2 == 0)
    {
        label maxCells1 = refineDict.lookup<label>("maxCells1");
        label maxCells2 = refineDict.lookup<label>("maxCells2");

        if (maxCells1 <= 0  || maxCells2 <= 0)
        {
            FatalErrorInFunction
                << "Illegal maximum number of cells " << maxCells1
                <<  " | " << maxCells2 << nl
                << "The maxCells setting in the dynamicMeshDict should"
                << " be > 0." << nl
                << exit(FatalError);
        }

        const label nBufferLayers1 =
            refineDict.lookup<label>("nBufferLayers1");
        const label nBufferLayers2 =
            refineDict.lookup<label>("nBufferLayers2");
```

Listing 4.2: New Initialisation phase

In general all the assignments in the initialisation phase need to be duplicated to account for the two fields that will be refined on in the simulation. In addition to this the checks for the well defined nature of the parameters need to be amended, so the the error prints out both the values to the output stream.

Since we have named the fields we shall refine with the `1` and `2` suffix respectively, the changes that will be made to the source code will be analogous for both fields. Hence for brevity, the changes for one of the fields will be explained and detailed, with the changes for the second field following suit. All that will be needed is a trivial change from the suffix `1` to the suffix `2`. The alterations to the code will be described according to whether they appear in the refinement or unrefinement phase of the `update` function.

4.3.3 Alterations for Refinement Phase

As with creating the key parameters for the initialisation phase of the `update` function, we put the suffix `1` on the key parameters, functions and data types in the refinement phase of the code. The full refinement phase of the `dynamicRefieFvMesh` class, which is contained in the `dynamicRefineFvMesh.C` file is found on lines 1375–1465 (Appendix B.2). The new refinement phase that is used in the `dynamicDualRefineFvMesh` class is given in Listing 4.3.

```
        // --- Field 1 --- //

        // Cells marked for refinement or otherwise protected from unrefinement.
        PackedBoolList refineCells1(nCells());

        label maxRefinement1 = 0;

        maxRefinement1 = selectRefineCandidates1(refineCells1, refineDict);

        if (globalData().nTotalCells() < maxCells1)
    {
        // Select subset of candidates. Take into account max allowable
        // cells, refinement level, protected cells.
        labelList cellsToRefine1
```

```
        (
            selectRefineCells
            (
                maxCells1,
                maxRefinement1,
                refineCells1
            )
        );

        label nCellsToRefine1 = returnReduce
        (
            cellsToRefine1.size(), sumOp<label>()
        );
        if (nCellsToRefine1 > 0)
        {
            // Refine/update mesh and map fields
            autoPtr<mapPolyMesh> map = refine(cellsToRefine1);

             // Update refineCells. Note that some of the marked ones have
             // not been refined due to constraints.
            {
                const labelList& cellMap = map().cellMap();
                const labelList& reverseCellMap = map().reverseCellMap();

                PackedBoolList newRefineCell(cellMap.size());

                forAll(cellMap, celli)
                {
                    label oldCelli = cellMap[celli];

                    if (oldCelli < 0)
                    {
                        newRefineCell.set(celli, 1);
                    }
                    else if (reverseCellMap[oldCelli] != celli)
                    {
                        newRefineCell.set(celli, 1);
                    }
                    else
                    {
                        newRefineCell.set(celli, refineCells1.get(oldCelli));
                    }
                }
                refineCells1.transfer(newRefineCell);
            }

                // Extend with a buffer layer to prevent neighbouring points
                // being unrefined.
                for (label i = 0; i < nBufferLayers1; i++)
                {
                    extendMarkedCells(refineCells1);
                }

                hasChanged = true;
            }
        }
```

Listing 4.3: New Refinement Phase

Table 4.1 shows all the parameters, labels, functions, etc that need to be changed and their new name in the section of code. It is observed that the `selectRefineCandidates` function has to be changed to `selectRefineCandidates1`; this is because the `selectRefineCandidates` function reads in the refinement parameters from the `dynamicMeshDict`. Since the parameter names have changed to account for the two fields that will be refined, then the `selectRefineCandidates` function needs to be adapted.

In order to do this, we can copy the `selectRefineCandidates` function defined on line 751 of

Table 4.1: List of the renamed parameters in the refinement phase of the `update` function with the type of the parameter also identified.

Original Name	New Name	Type
`refineCells`	`refineCells1`	`PackedBoolList`
`maxRefinement`	`maxRefinement1`	`label`
`selectRefineCandidates`	`selectRefineCandidates1`	`function`
`maxCells`	`maxCells1`	`label`
`nCellsToRefine`	`nCellsToRefine1`	`label`
`cellsToRefine`	`cellsToRefine1`	`labelList`
`nBufferLayers`	`nBufferLayers1`	`label`

`dynamicRefineFvMesh.C` (See Appendix B.2) and paste it into the `dynamicDualRefineFvMesh.C` file at line 806. The original set of `selectRefineCandidates` functions are found on lines 684 – 804 in the `dynamicRefineFvMesh.C` file (Appendix B.2) We shall create the `selectRefineCandidates1` function in order to read the parameters that will govern the mesh refinement for the first field in the CFD simulation. After pasting `selectRefineCandidates` function from line 751 of the `dynamicRefineFvMesh.C` file, to line 806 of the `dynamicDualRefineFvMesh.C` file all that needs to be changed is the names of the parameters read from the `dynamicMeshDict` in order to ensure the correct parameters are read in. First the name of the function must be changed to reflect the new class that the function is a member of, and also the change in name to signal that this function will read the data from the first refinement field.

```
806 Foam::scalar Foam::dynamicDualRefineFvMesh::selectRefineCandidates1
```

After this the field name

```
812     const word fieldName(refineDict.lookup("field1"));
```

`lowerRefineLevel`, `upperRefineLevel`

```
816     const scalar lowerRefineLevel =
817         refineDict.lookup<scalar>("lowerRefineLevel1");
818     const scalar upperRefineLevel =
819         refineDict.lookup<scalar>("upperRefineLevel1");
```

and `maxRefinement`

```
821     const label maxRefinement = refineDict.lookup<label>("maxRefinement1");
```

are all updated to ensure that the proper fields are read from the `dynamicMeshDict` that is stored locally as the `refineDict` dictionary in the `update` function. After this the `selectRefineCandidates` function can be unaltered since the refinement criteria and procedure for the field is the same as in the existing `dynamicRefineFvMesh` class. The newly created `selectRefineCandidates1` function is presented in Listing 4.4.

```
806 Foam::scalar Foam::dynamicDualRefineFvMesh::selectRefineCandidates1
807 (
808     PackedBoolList& candidateCells,
809     const dictionary& refineDict
810 ) const
811 {
812     const word fieldName(refineDict.lookup("field1"));
813
814     const volScalarField& vFld = lookupObject<volScalarField>(fieldName);
815
816     const scalar lowerRefineLevel =
817         refineDict.lookup<scalar>("lowerRefineLevel1");
818     const scalar upperRefineLevel =
819         refineDict.lookup<scalar>("upperRefineLevel1");
820
```

```
    const label maxRefinement = refineDict.lookup<label>("maxRefinement1");

    if (maxRefinement <= 0)
    {
        FatalErrorInFunction
            << "Illegal maximum refinement level " << maxRefinement << nl
            << "The maxCells setting in the dynamicMeshDict should"
            << " be > 0." << nl
            << exit(FatalError);
    }

    // Determine candidates for refinement (looking at field only)
    selectRefineCandidates
    (
        candidateCells,
        lowerRefineLevel,
        upperRefineLevel,
        maxRefinement,
        vFld
    );

    return maxRefinement;
}
```

Listing 4.4: New `selectRefineCandidates1` function

Since we have defined a new member function `selectRefineCandidates1`, we must ensure it is declared in the `dynamicDualRefineFvMesh.H` file. We are able to copy the declaration of the `selectRefineCandidates` function from the `dynamicRefineFvMesh.H` file for this, since the new function takes in the same input parameters as the old function, then all that must change is the name.

The code added to the `dynamicDualRefineFvMesh.H` file is give in Listing 4.5

```
            virtual scalar selectRefineCandidates1
            (
                PackedBoolList& candidateCell,
                const dictionary& refineDict
            ) const;
```

Listing 4.5: Declaration of `selectRefineCandidates1` function

4.3.4 Alterations for Unrefinement Phase

The unrefinement phase of the `update` function will be adapted in a similar manner to the refinement phase of the code as presented in the previous section. Similarly, the changes here are analogous to the changes that are made the second field, in which the user will change the suffix `1` to `2`. In addition, the `dynamicDualRefineFvMesh` class will be extended to allow for the unrefinement of a field, if the value of of the field chosen to refine the mesh on is larger then a specified value. The original unrefinement phase of the `update` function is found in lines 1467–1518 of the `dynamicRefineFvMesh.C` file (See Appendix B.2). The new refinement phase for one of the fields in the simulation for the `dynamicDualRefineFvMesh` class is given in Listing 4.6.

```
        boolList unrefineCandidates1(nPoints(), true);

        selectUnrefineCandidates1
        (
            unrefineCandidates1,
            refineDict
        );

        {

            // Select unrefineable points that are not marked in refineCells
```

```
            labelList pointsToUnrefine1
            (
                selectUnrefinePoints
                (
                    refineCells1,
                    unrefineCandidates1
                )
            );

            label nSplitPoints1 = returnReduce
            (
                pointsToUnrefine1.size(),
                sumOp<label>()
            );

            if (nSplitPoints1 > 0)
            {
                // Refine/update mesh
                unrefine(pointsToUnrefine1);
                hasChanged = true;
            }
        }
```

Listing 4.6: New Unrefinement Phase

We can notice again, that all that changes are the names of some of the key parameters and variables in the unrefinement phase. A list of the changed names in the unrefinement part of the function is given in Table 4.2. In similar fashion to the refinement phase, the key adaptation to the unrefinement phase, is the change to the `selectUnrefineCandidates` function. Once again we change this to handle the reading of the changed parameters name, but crucially we shall change this to add the capabilities of more flexible unrefinement.

4.4 New `selectUnrefineCandidates` function

The `selectUnrefineCandidates` functions are on lines 894–983 of the `dynamicRefineFvMesh.C` file (Appendix B.2). We shall add flexibility to this function to allow the user to unrefine the mesh if the field selected to control the mesh refinements is greater than a user defined threshold value. This threshold value will be named `upperUnrefineLevel`, with the previous variable `unrefineLevel` now being named `lowerUnrefineLevel`. At present, it is possible to unrefine the mesh only if the field chosen to control the mesh refinements in the cells around a point is less than the `unrefineLevel` parameter. The `upperUnrefineLevel` parameter will be introduced in order to achieve the unrefinement of a the mesh if the field chosen to control the mesh in the cells around a point is greater than the `upperUnrefineLevel` parameter.

Recall that when unrefining the computational mesh, we consider removing points in the mesh to make larger cells. In order to implement this we need to find the value of the refinement field in the cells that surround a point in the mesh. We can use the `pointCells()` function in order to loop

Table 4.2: List of the renamed parameters in the unrefinement phase of the `update` function with the type of the parameter also identified.

Old Name	New Name	Type
`unrefineCandidates`	`unrefineCandidates1`	`boolList`
`selectUnrefineCandidates`	`selectUnrefineCandidates1`	`function`
`pointsToUnrefine`	`pointsToUnrefine1`	`labelList`
`nSplitPoints`	`nSplitPoints1`	`label`
`refineCells`	`refineCells1`	`PackedBoolList`
`unrefineLevel`	`lowerUnrefineLevel`	`label`

through all the cells that border a point in the computational mesh in order to find the field value of those cells. The code to do this is given in Listing 4.7

```
forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar minVal = great;
            forAll(pCells, i)
            {
                minVal = min(minVal, vFld[pCells[i]]);
            }
```

Listing 4.7: Code for finding the minimum value of a certain volume field in the cells surrounding a point in the computational mesh.

In this code we loop over all points in the computational mesh, for each point we store a list of all the cells that border that point in the `pCells` variable. Then an arbitrary value for the `minVal` parameter is created, this is set to be `great` since in the coming for–loop this value will be ensured to be overwritten, because in a stable simulation all values of any volume field should be finite valued. Next we loop over all the border cells of that point and find the minimum value of that field in the cells around that points. Hence `minVal` ends as the minimum of all the cells around a point in the computational mesh. In order the unrefine the mesh, all points such that the `minVal` is greater than the `upperUnRefineLevel` are marked as candidates to unrefine.

```
unrefineCandidates[pointi] =
    unrefineCandidates[pointi] && minVal > upperUnrefineLevel
```

To make the selection of the refinement flexible, the `selectUnrefeinCandidates1` function of the `dyanamicDualRefineFvMesh` class will be adapted so that the user can other specify one or both of the `lowerUnrefineLevel` and `upperUnrefineLevel`. To introduce this change in the `selectUnrefineCandidates1` function, the code given in Listing 4.7 needs to be added to the function. In addition to this a series of logical statements can be added to the function so the either the mesh is unrefined in places such that `field` < `lowerUnrefineLevel` and `upperUnrefineLevel` < `field` or `upperUnrefineLevel` < `field` or `field` < `lowerUnrefineLevel`, depending on which of `lowerUnrefineLevel` and `upperUnrefineLevel` are defined in the `dynamicMeshDict`. The new `selectUnrefineCandidates` function is given in Listing 4.8. We can see that the handling of different unrefinement criteria is selected using the multiple `if` statements, depending on the entries that appear in the `dynamicMeshDict`.

```
void Foam::dynamicDualRefineFvMesh::selectUnrefineCandidates1
(
    boolList& unrefineCandidates,
    const dictionary& refineDict
) const
{
    if (refineDict.found("lowerUnrefineLevel1")
     && refineDict.found("upperUnrefineLevel1"))
    {
        const word fieldName(refineDict.lookup("field1"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar lowerUnrefineLevel =
            refineDict.lookup<scalar>("lowerUnrefineLevel1");

        const scalar upperUnrefineLevel =
            refineDict.lookup<scalar>("upperUnrefineLevel1");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];
```

```

            scalar maxVal = -great;
            forAll(pCells, i)
            {
                maxVal = max(maxVal, vFld[pCells[i]]);
            }

            scalar minVal = great;
            forAll(pCells, i)
            {
                minVal = min(minVal, vFld[pCells[i]]);
            }

            unrefineCandidates[pointi] =
                (unrefineCandidates[pointi] && maxVal < lowerUnrefineLevel)
             || (unrefineCandidates[pointi] && minVal > upperUnrefineLevel);
        }
    }

    if (refineDict.found("lowerUnrefineLevel1")
     && !refineDict.found("upperUnrefineLevel1"))

    {
        const word fieldName(refineDict.lookup("field1"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar lowerUnrefineLevel =
            refineDict.lookup<scalar>("lowerUnrefineLevel1");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar maxVal = -great;
            forAll(pCells, i)
            {
                maxVal = max(maxVal, vFld[pCells[i]]);
            }

            unrefineCandidates[pointi] =
                (unrefineCandidates[pointi] && maxVal < lowerUnrefineLevel);
        }
    }

    if (!refineDict.found("lowerUnrefineLevel1")
     && refineDict.found("upperUnrefineLevel1"))
    {
        const word fieldName(refineDict.lookup("field1"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar upperUnrefineLevel =
            refineDict.lookup<scalar>("upperUnrefineLevel1");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar minVal = great;
            forAll(pCells, i)
            {
                minVal = min(minVal, vFld[pCells[i]]);
            }
```

```
                unrefineCandidates[pointi] =
            (unrefineCandidates[pointi] && minVal > upperUnrefineLevel);
            }
    }
}


void Foam::dynamicDualRefineFvMesh::selectUnrefineCandidates2
(
    boolList& unrefineCandidates,
    const dictionary& refineDict
) const
{
    if (refineDict.found("lowerUnrefineLevel2")
     && refineDict.found("upperUnrefineLevel2"))
    {
        const word fieldName(refineDict.lookup("field2"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar lowerUnrefineLevel =
            refineDict.lookup<scalar>("lowerUnrefineLevel2");

        const scalar upperUnrefineLevel =
            refineDict.lookup<scalar>("upperUnrefineLevel2");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar maxVal = -great;
            forAll(pCells, i)
            {
                maxVal = max(maxVal, vFld[pCells[i]]);
            }

            scalar minVal = great;
            forAll(pCells, i)
            {
                minVal = min(minVal, vFld[pCells[i]]);
            }

            unrefineCandidates[pointi] =
                (unrefineCandidates[pointi] && maxVal < lowerUnrefineLevel)
             || (unrefineCandidates[pointi] && minVal > upperUnrefineLevel);
        }
    }
    if (refineDict.found("lowerUnrefineLevel2")
     && !refineDict.found("upperUnrefineLevel2"))

    {
        const word fieldName(refineDict.lookup("field2"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar lowerUnrefineLevel =
            refineDict.lookup<scalar>("lowerUnrefineLevel2");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar maxVal = -great;
```

```
            forAll(pCells, i)
            {
                maxVal = max(maxVal, vFld[pCells[i]]);
            }

            unrefineCandidates[pointi] =
                (unrefineCandidates[pointi] && maxVal < lowerUnrefineLevel);
        }
    }

    if (!refineDict.found("lowerUnrefineLevel2")
     && refineDict.found("upperUnrefineLevel2"))
    {
        const word fieldName(refineDict.lookup("field2"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar upperUnrefineLevel =
            refineDict.lookup<scalar>("upperUnrefineLevel2");

        forAll(pointCells(), pointi)
        {
            const labelList& pCells = pointCells()[pointi];

            scalar minVal = great;
            forAll(pCells, i)
            {
                minVal = min(minVal, vFld[pCells[i]]);
            }

            unrefineCandidates[pointi] =
                (unrefineCandidates[pointi] && minVal > upperUnrefineLevel);
        }
    }
}
```

Listing 4.8: New `selectUnrefineCandidates` function

Again, since we have defined a new member function `selectUnrefineCandidates1` in the class `dynamicDualRefineFvMesh` then we must ensure it is declared in the `dynamicDualRefineFvMesh.H` file. For this we are able to copy the declaration of the `selectUnrefineCandidates` function from the `dynamicRefineFvMesh.H` file. Since the new function takes in the same input parameters as the old function, then all that must change is the name. The code added to the `dynamicDualRefineFvMesh.H` file is give in Listing 4.9

```
            void selectUnrefineCandidates1
            (
                boolList& unrefineCandidates,
                const dictionary& refineDict
            ) const;
```

Listing 4.9: Declaration of `selectUnrefineCandidates1` function

4.5 Summary

The adaptations completed in this section are to create one field of refinement for the AMR algorithm in OpenFOAM. As stated in the introduction to this chapter, the same procedure in creating the refinement code for `field1` is applied for `field2`. All that changes is the suffix `1` to `2` in all the variables listed in Table 4.1 and Table 4.2. Since this is a trivial matter, the details will not be stated explicitly. The full `dynamicDualRefineFvMesh` class code is provided in the supplementary material of the report files. In order to compile the library from the provided materials without going through the changes to the source code explained in this chapter, the user can unzip the class from the provided materials and use the `Allwmake` script to compile the library in a directory of their choosing. At this point we are in a position to test the `dynamicDualRefineFvMesh` class library, and apply it to our problem of choice to demonstrate its capabilities.

Chapter 5

Using dynamicDualRefineFvMesh

5.1 Modifying the tutorial

We shall use the damBreakWithObstacle tutorial case in order to demonstrate the capabilities of the dynamicDualRefineFvMesh class created in Chapter 4. To use the dynamicDualRefineFvMesh class, we shall return to the test case created in Section 4.2.1. If the dynamicDualRefineFvMesh library is intended to be used on any other case, then the library must be linked in the controlDict file as illustrated in Section 4.2.1. After the library has been linked to the case, all that is left to do is to add the entries for field 1 and field 2 in the dynamicMeshDict.

5.1.1 The dynamicMeshDict

The changes made to the dynamicMeshDict are such that most entries in the dictionary must be duplicated in order to specify the refinement parameters for the first and second refinement fields (field1 and field2). In addition to this the lowerUnrefineLevel and upperUnrefineLevel can be added to the dictionary for each phase due to the new unrefine functionality added to the class.

An example dynamicMeshDict used for the demonstration of the two field refinement capabilities of dynamicDualRefineFvMesh is given in Listing 5.1

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  9
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    location    "constant";
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh   dynamicDualRefineFvMesh;

// --- Field 1 -- Interface --- //
// How often to refine
refineInterval1  1;

// Field to be refinement on
field1           alpha.water;

// Refine field in between lower..upper
```

```
lowerRefineLevel1 0.001;
upperRefineLevel1 0.999;

// If value < unrefineLevel unrefine
lowerUnrefineLevel1   0;
upperUnrefineLevel1   0.4;

// Have slower than 2:1 refinement
nBufferLayers1   2;

// Refine cells only up to maxRefinement levels
maxRefinement1   2;

// Stop refinement if maxCells reached
maxCells1        200000;


// --- Field 2 -- Bulk Water --- //
// How often to refine
refineInterval2  1;

// Field to be refinement on
field2           alpha.water;

// Refine field in between lower..upper
lowerRefineLevel2 0.4;
upperRefineLevel2 1.1;

// If value < unrefineLevel unrefine
lowerUnrefineLevel2   0.5;

// Have slower than 2:1 refinement
nBufferLayers2   1;

// Refine cells only up to maxRefinement levels
maxRefinement2   1;

// Stop refinement if maxCells reached
maxCells2        200000;

// Flux field and corresponding velocity field. Fluxes on changed
// faces get recalculated by interpolating the velocity. Use 'none'
// on surfaceScalarFields that do not need to be reinterpolated.
correctFluxes
(
    (phi none)
    (nHatf none)
    (rhoPhi none)
    (alphaPhi0.water none)
    (ghf none)
);

// Write the refinement level as a volScalarField
dumpLevel       false;
```

Listing 5.1: `dynamicMeshDict` example for `dynamicDualRefineFvMesh`

Hence it is seen that most of the original entries of the `dynamicMeshDict` are duplicated to account for the two fields that will be refined on in the simulation.

5.1.2 Using `dynamicDualRefineFvMesh` to generate independent bulk and interface mesh refinement

The motivation of this project is to create a mesh class such that one phase within a two phase flow can be refined, but also an independent refinement level can be generated at the interface of the flow. In this section we shall demonstrate how the `dynamicDualRefineFvMesh` class can be used to do this. We shall use `field1` to refine the air–water interface in the `testDamBreak` simulation, and `field2` to refine/unrefine the bulk of the water phase. The `dynamicMeshDict` shown in Listing 5.1 shows an example of the parameter settings that can be used to conduct the mesh refinement desired.

5.1.2.1 Interface Refinement

Listing 5.2, shows the parameters used in the `dynamicMeshDict` In order to refine the air–water interface of the `testDamBreak` simulation using `dynamicDualRefineFvMesh`.

```
// --- Field 1 -- Interface --- //
// How often to refine
refineInterval1  1;

// Field to be refinement on
field1           alpha.water;

// Refine field in between lower..upper
lowerRefineLevel1 0.001;
upperRefineLevel1 0.999;

// If value < lowerUnrefineLevel unrefine
lowerUnrefineLevel1   0;

// If value > upperUnrefineLevel unrefine
upperUnrefineLevel1   0.4;

// Have slower than 2:1 refinement
nBufferLayers1   2;

// Refine cells only up to maxRefinement levels
maxRefinement1   2;

// Stop refinement if maxCells reached
maxCells1        200000;
```

Listing 5.2: Interface Field Refinement Paramters

In order to refine the mesh around the interface, we shall use the standard entries that were present in the original `damBreakWithObstacle` case. We use the `alpha.water` field, with `lowerRefineLevel` and `upperRefineLevel` set to 0.001 and 0.999 respectively to ensure refinement on all cells that contain the multiphase interface. In addition to this we shall use the new `lowerUnrefineLevel` and `upperRefinementLevel` functionality. To ensure that any cells that lie in bulk of the two phases in the simulation are unrefined from the interface refinement level. To ensure the interface is more refined than the bulk we set the `maxRefinement` parameter for the interface to 2, and the `nBufferLayers` parameter will be set to 2 in order emphasize the mesh resolution around the interface.

5.1.2.2 Bulk Refinement

Listing 5.3 shows the parameters we use to carry out the wanted refinement in the bulk of the `alpha.water` phase.

```
// --- Field 2 -- Bulk Water --- //
// How often to refine
refineInterval2  1;

// Field to be refinement on
field2           alpha.water;

// Refine field in between lower..upper
lowerRefineLevel2 0.4;
upperRefineLevel2 1.1;

```

```
// If value < unrefineLevel unrefine
lowerUnrefineLevel2   0.001;

// Have slower than 2:1 refinement
nBufferLayers2   2;

// Refine cells only up to maxRefinement levels
maxRefinement2   1;

// Stop refinement if maxCells reached
maxCells2        200000;
```

Listing 5.3: Bulk Field Refinement Paramters

To refine the mesh around in the bulk, we shall use the `alpha.water` field, with `lowerRefineLevel` and `upperRefineLevel` set to 0.999 and 1.1 respectively. Recall the `alpha.water` phase should be bounded between 0 and 1, with the cells containing the `alpha.water` having value 1. We increase the limit to 1.1 to account for the numerical error that is introduced in the Finite Volume discretisation. This way all bulk water cells will be accounted for. In addition to this we shall solely use the `lowerUnrefineLevel` functionality to control the unrefinement of the bulk phase. To ensure that no cells that contain only the air phase in the simulation are refined the `lowerUnrefineLevel` is set to 0.001. Since we desire a level of mesh refinement higher than the background mesh that is found in the air phase, but lower than the level on the interface we set the `maxRefinement` parameter for the bulk water phase to 1. Again the `nBufferLayers` parameter is also to set to 2 in order to more easily visualise the transition regions of the mesh.

5.2 Results

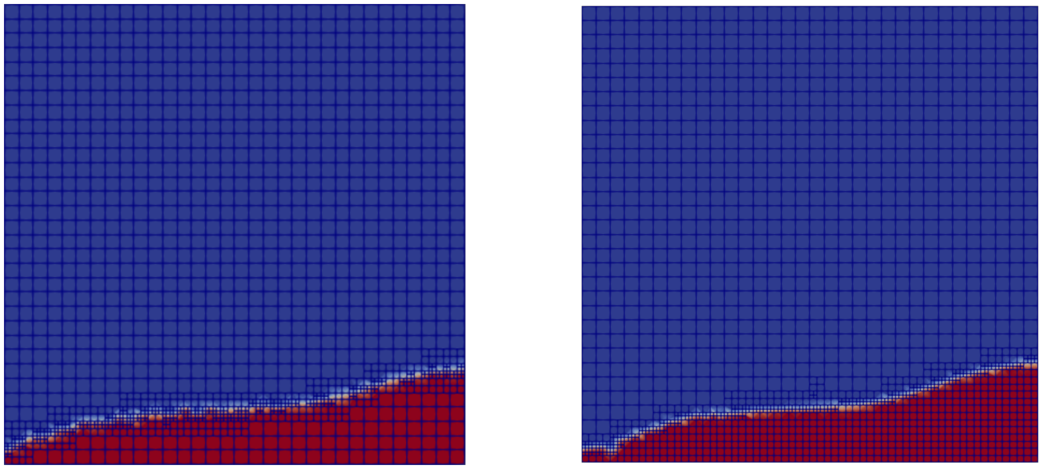
To run the simulation with the `dynamicDualRefineFvMesh`, we shall execute

```
blockMesh
setFields
interFoam
```

It is noted here that we do not use the `Allrun` script that is provided with the `damBreakWithObstacle` tutorial, for one because some amendments to this script are needed to ensure it calls the user created library, but also since the evolution of the water over the obstacle is not necessarily of interest in the report, but demonstrating the changes to the mesh are. In its current state, the tutorial case crashes around 1.1 s into the simulation, the nature of this is not fully understood at present. The current theory is that there is some interference between the refinement procedures of the two fields used in the simulation. Changing the refinement parameters in the `dyanmicMeshDict` causes a difference in the run–time of the simulation before it crashes. Further work would be needed to understand if this is a case specific problem and determine a solution. Once the simulation results can still be viewed in Paraview. Figure 5.1 shows a comparison of the mesh at time t = 0.4s in the simulation, using the same visualisation view as in Chapter 2 when using the `dynamicRefineFvMesh` and `dynamicDualRefineFvMesh` classes to handle to the AMR in the simulation. From Figure 5.1 we can see that indeed the mesh refinement we desired has been generated. We find the initial mesh is preserved in the air phase, one level of mesh refinement in the water phase and two levels of refinement at the air–water interface.

5.3 Conclusion

Figure 5.1 shows it is possible with a very small adaptation of the OpenFOAM source code to refine on multiple evolving fields in a CFD simulation. In addition to this we have shown it is possible to employ this adaptation to the source code, such that any two fields can be refined on in the simulation. The reader is encouraged to experiment with the `dynamicDualRefineFvMesh` library in other tutorial cases to explore the limits of its capabilities.



(a) `dynamicRefineFvMesh` (b) `dynamicDualRefineFvMesh`

Figure 5.1: Comparison of the mesh on the +X face of the simulation for the `damBreakWithObstacle` tutorial at time $t = 0.4\,\mathrm{s}$ using either `dynamicRefineFvMesh` and `dynamicDualRefineFvMesh`.

It is noted that the implementation of this library is not the most efficient or general, but the purpose of this report was to create a proof of concept library to understand how to create a method of generating two–field refinement in an OpenFOAM simulation. The library can be made more effective and flexible by using dictionary inputs like the `refinementRegions` functionality in OpenFOAM 9 does for static regions of mesh refinement. In this was the implementation can be made more general, by allowing multiple refinement fields, which will be handled by a loop in the source code. This would be the main direction of future work to extend this library.

Bibliography

[1] A. Kosters, "Dynamic mesh refinement in dieselFoam." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/anneKoesters/anneKoestersReport.pdf.

[2] A. Nygren, "Adaptive mesh refinement with a moving mesh using sprayDyMFoam." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2015/AndreasNygren/Tutorial_SprayDyMFoam.pdf.

[3] D. Lindblad, "Implementation and run-time mesh refinement for the $k - -\omega$ SST DES turbulence model when applied to airfoils.." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2013/DanielLindblad/k-Omega-SST-DES-Report.pdf.

[4] B. Eltard-Larsen, "How to make a dynamicMotionRefineFvMesh class." http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2015/BjarkeEltard-Larsen/dynamicMotionRefineFvMesh_revised.pdf.

[5] T. Holzmann, "dynamicRefineFvMesh with two regions." https://www.cfd-online.com/Forums/openfoam-community-contributions/162715-dynamicrefinefvmesh-two-regions.html.

[6] T. Holzmann, "Holzmann cfd." http://www.holzmann-cfd.de/index.php/en/development.

[7] D. Rettenmaier, D. Deising, Y. Ouedraogo, E. Gjonaj, H. De Gersem, D. Bothe, C. Tropea, and H. Marschall, "Load balanced 2d and 3d adaptive mesh refinement in openfoam," *SoftwareX*, vol. 10, p. 100317, 2019.

[8] J. Castrejón-Pita, K. Kubiak, A. Castrejón-Pita, M. Wilson, and I. Hutchings, "Mixing and internal dynamics of droplets impacting and coalescing on a solid surface," *Physical Review E*, vol. 88, no. 2, p. 023023, 2013.

[9] P. Kröber, J. T. Delaney, J. Perelaer, and U. S. Schubert, "Reactive inkjet printing of polyurethanes," *Journal of Materials Chemistry*, vol. 19, no. 29, pp. 5234–5238, 2009.

[10] S.-I. Yeh, H.-J. Sheen, and J.-T. Yang, "Chemical reaction and mixing inside a coalesced droplet after a head-on collision," *Microfluidics and Nanofluidics*, vol. 18, no. 5, pp. 1355–1363, 2015.

[11] J. Eggers, J. R. Lister, and H. A. Stone, "Coalescence of liquid drops," *Journal of Fluid Mechanics*, vol. 401, pp. 293–310, 1999.

[12] J. Jin, C. H. Ooi, D. V. Dao, and N.-T. Nguyen, "Coalescence processes of droplets and liquid marbles," *Micromachines*, vol. 8, no. 11, p. 336, 2017.

[13] M. Brik, S. Harmand, I. Zaaroura, and A. Saboni, "Experimental and numerical study for the coalescence dynamics of vertically aligned water drops in oil," *Langmuir*, vol. 37, no. 10, pp. 3139–3147, 2021.

Study questions

1. Which field should you use in the `damBreakWithTutorial` simulation to refine the air–water interface?
2. What setting should be used for `refinetInterval` to ensure mesh refinement only takes place every 5 time steps?
3. Which keyword can be used to in the `correctFluxes` table for fluxes that do not need to be re–interpolated?
4. In what file is the `mesh` object created, and what is this object and instance of?
5. What type of fields can be refined on currently in OpenFOAM using the `dynamicRefineFvMesh` class? How would other fields be refined on?
6. What is the name of the local dictionary that the `dynamicMeshDict` is stored as during mesh refinement?
7. How do you find the maximum value of a field in the cells around a point in the CFD mesh?
8. What happens if the number of cells marked to be refined will cause the number of cells in the simulation to exceed maxCells— and how In the code is it estimated if this will occur?
9. In the unrefinement phase of the AMR code, why are points considered and not cells?

Appendix A

Dictionaries

A.1 damBreakWithObstacle dynamicMeshDict

dynamicMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  9
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    location    "constant";
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh   dynamicRefineFvMesh;

// How often to refine
refineInterval  1;

// Field to be refinement on
field           alpha.water;

// Refine field in between lower..upper
lowerRefineLevel 0.001;
upperRefineLevel 0.999;

// If value < unrefineLevel unrefine
unrefineLevel   10;

// Have slower than 2:1 refinement
nBufferLayers   1;

// Refine cells only up to maxRefinement levels
maxRefinement   2;

// Stop refinement if maxCells reached
maxCells        200000;

// Flux field and corresponding velocity field. Fluxes on changed
// faces get recalculated by interpolating the velocity. Use 'none'
// on surfaceScalarFields that do not need to be reinterpolated.
correctFluxes
```

```
(
    (phi none)
    (nHatf none)
    (rhoPhi none)
    (alphaPhi0.water none)
    (ghf none)
);

// Write the refinement level as a volScalarField
dumpLevel       true;


// ************************************************************************* //
```

Appendix B

Source Codes

B.1 interFoam.C

interFoam.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2021 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    interFoam

Description
    Solver for 2 incompressible, isothermal immiscible fluids using a VOF
    (volume of fluid) phase-fraction based interface capturing approach,
    with optional mesh motion and mesh topology changes including adaptive
    re-meshing.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "CMULES.H"
#include "EulerDdtScheme.H"
#include "localEulerDdtScheme.H"
#include "CrankNicolsonDdtScheme.H"
#include "subCycle.H"
#include "immiscibleIncompressibleTwoPhaseMixture.H"
#include "noPhaseChange.H"
#include "kinematicMomentumTransportModel.H"
```

```
#include "pimpleControl.H"
#include "pressureReference.H"
#include "fvModels.H"
#include "fvConstraints.H"
#include "CorrectPhi.H"
#include "fvcSmooth.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"
    #include "createFields.H"
    #include "createFieldRefs.H"
    #include "createAlphaFluxes.H"
    #include "initCorrectPhi.H"
    #include "createUfIfPresent.H"

    turbulence->validate();

    if (!LTS)
    {
        #include "CourantNo.H"
        #include "setInitialDeltaT.H"
    }

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    Info<< "\nStarting time loop\n" << endl;

    while (pimple.run(runTime))
    {
        #include "readDyMControls.H"

        if (LTS)
        {
            #include "setRDeltaT.H"
        }
        else
        {
            #include "CourantNo.H"
            #include "alphaCourantNo.H"
            #include "setDeltaT.H"
        }

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            if (pimple.firstPimpleIter() || moveMeshOuterCorrectors)
            {
                // Store divU from the previous mesh so that it can be mapped
                // and used in correctPhi to ensure the corrected phi has the
                // same divergence
                tmp<volScalarField> divU;

                if
                (
                    correctPhi
                 && !isType<twoPhaseChangeModels::noPhaseChange>(phaseChange)
```

```
            )
            {
                // Construct and register divU for mapping
                divU = new volScalarField
                (
                    "divU0",
                    fvc::div(fvc::absolute(phi, U))
                );
            }

            fvModels.preUpdateMesh();

            mesh.update();

            if (mesh.changing())
            {
                // Do not apply previous time-step mesh compression flux
                // if the mesh topology changed
                if (mesh.topoChanging())
                {
                    talphaPhi1Corr0.clear();
                }

                gh = (g & mesh.C()) - ghRef;
                ghf = (g & mesh.Cf()) - ghRef;

                MRF.update();

                if (correctPhi)
                {
                    #include "correctPhi.H"
                }

                mixture.correct();

                if (checkMeshCourantNo)
                {
                    #include "meshCourantNo.H"
                }
            }

            divU.clear();
        }

        fvModels.correct();

        surfaceScalarField rhoPhi
        (
            IOobject
            (
                "rhoPhi",
                runTime.timeName(),
                mesh
            ),
            mesh,
            dimensionedScalar(dimMass/dimTime, 0)
        );

        #include "alphaControls.H"
        #include "alphaEqnSubCycle.H"

        mixture.correct();

        #include "UEqn.H"

        // --- Pressure corrector loop
        while (pimple.correct())
        {
```

```
                #include "pEqn.H"
            }

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


// ************************************************************************* //
```

B.2 dynamicRefineFvMesh.C

dynamicRefineFvMesh.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2021 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "dynamicRefineFvMesh.H"
#include "surfaceInterpolate.H"
#include "polyTopoChange.H"
#include "syncTools.H"
#include "pointFields.H"
#include "sigFpe.H"
#include "cellSet.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
```

```
{
    defineTypeNameAndDebug(dynamicRefineFvMesh, 0);
    addToRunTimeSelectionTable(dynamicFvMesh, dynamicRefineFvMesh, IOobject);
}

// * * * * * * * * * * * * Protected Member Functions  * * * * * * * * * * * //

Foam::label Foam::dynamicRefineFvMesh::count
(
    const PackedBoolList& l,
    const unsigned int val
)
{
    label n = 0;
    forAll(l, i)
    {
        if (l.get(i) == val)
        {
            n++;
        }

        // Debug also serves to get-around Clang compiler trying to optimise
        // out this forAll loop under O3 optimisation
        if (debug)
        {
            Info<< "n=" << n << endl;
        }
    }

    return n;
}


void Foam::dynamicRefineFvMesh::calculateProtectedCells
(
    PackedBoolList& unrefineableCells
) const
{
    if (protectedCells_.empty())
    {
        unrefineableCells.clear();
        return;
    }

    const labelList& cellLevel = meshCutter_.cellLevel();

    unrefineableCells = protectedCells_;

    // Get neighbouring cell level
    labelList neiLevel(nFaces()-nInternalFaces());

    for (label facei = nInternalFaces(); facei < nFaces(); facei++)
    {
        neiLevel[facei-nInternalFaces()] = cellLevel[faceOwner()[facei]];
    }
    syncTools::swapBoundaryFaceList(*this, neiLevel);


    while (true)
    {
        // Pick up faces on border of protected cells
        boolList seedFace(nFaces(), false);

        forAll(faceNeighbour(), facei)
        {
            label own = faceOwner()[facei];
            bool ownProtected = unrefineableCells.get(own);
            label nei = faceNeighbour()[facei];
```

```
            bool neiProtected = unrefineableCells.get(nei);

            if (ownProtected && (cellLevel[nei] > cellLevel[own]))
            {
                seedFace[facei] = true;
            }
            else if (neiProtected && (cellLevel[own] > cellLevel[nei]))
            {
                seedFace[facei] = true;
            }
        }
        for (label facei = nInternalFaces(); facei < nFaces(); facei++)
        {
            label own = faceOwner()[facei];
            bool ownProtected = unrefineableCells.get(own);
            if
            (
                ownProtected
             && (neiLevel[facei-nInternalFaces()] > cellLevel[own])
            )
            {
                seedFace[facei] = true;
            }
        }

        syncTools::syncFaceList(*this, seedFace, orEqOp<bool>());


        // Extend unrefineableCells
        bool hasExtended = false;

        for (label facei = 0; facei < nInternalFaces(); facei++)
        {
            if (seedFace[facei])
            {
                label own = faceOwner()[facei];
                if (unrefineableCells.get(own) == 0)
                {
                    unrefineableCells.set(own, 1);
                    hasExtended = true;
                }

                label nei = faceNeighbour()[facei];
                if (unrefineableCells.get(nei) == 0)
                {
                    unrefineableCells.set(nei, 1);
                    hasExtended = true;
                }
            }
        }
        for (label facei = nInternalFaces(); facei < nFaces(); facei++)
        {
            if (seedFace[facei])
            {
                label own = faceOwner()[facei];
                if (unrefineableCells.get(own) == 0)
                {
                    unrefineableCells.set(own, 1);
                    hasExtended = true;
                }
            }
        }

        if (!returnReduce(hasExtended, orOp<bool>()))
        {
            break;
        }
    }
```

```
}


void Foam::dynamicRefineFvMesh::readDict()
{
    const dictionary refineDict
    (
        dynamicMeshDict().optionalSubDict(typeName + "Coeffs")
    );

    List<Pair<word>> fluxVelocities = List<Pair<word>>
    (
        refineDict.lookup("correctFluxes")
    );
    // Rework into hashtable.
    correctFluxes_.resize(fluxVelocities.size());
    forAll(fluxVelocities, i)
    {
        correctFluxes_.insert(fluxVelocities[i][0], fluxVelocities[i][1]);
    }

    dumpLevel_ = Switch(refineDict.lookup("dumpLevel"));
}


// Refines cells, maps fields and recalculates (an approximate) flux
Foam::autoPtr<Foam::mapPolyMesh>
Foam::dynamicRefineFvMesh::refine
(
    const labelList& cellsToRefine
)
{
    // Mesh changing engine.
    polyTopoChange meshMod(*this);

    // Play refinement commands into mesh changer.
    meshCutter_.setRefinement(cellsToRefine, meshMod);

    // Create mesh (with inflation), return map from old to new mesh.
    // autoPtr<mapPolyMesh> map = meshMod.changeMesh(*this, true);
    autoPtr<mapPolyMesh> map = meshMod.changeMesh(*this, false);

    Info<< "Refined from "
        << returnReduce(map().nOldCells(), sumOp<label>())
        << " to " << globalData().nTotalCells() << " cells." << endl;

    if (debug)
    {
        // Check map.
        for (label facei = 0; facei < nInternalFaces(); facei++)
        {
            label oldFacei = map().faceMap()[facei];

            if (oldFacei >= nInternalFaces())
            {
                FatalErrorInFunction
                    << "New internal face:" << facei
                    << " fc:" << faceCentres()[facei]
                    << " originates from boundary oldFace:" << oldFacei
                    << abort(FatalError);
            }
        }
    }

    // Update fields
    updateMesh(map);

    // Correct the flux for modified/added faces. All the faces which only
```

```
    // have been renumbered will already have been handled by the mapping.
    {
        const labelList& faceMap = map().faceMap();
        const labelList& reverseFaceMap = map().reverseFaceMap();

        // Storage for any master faces. These will be the original faces
        // on the coarse cell that get split into four (or rather the
        // master face gets modified and three faces get added from the master)
        labelHashSet masterFaces(4*cellsToRefine.size());

        forAll(faceMap, facei)
        {
            label oldFacei = faceMap[facei];

            if (oldFacei >= 0)
            {
                label masterFacei = reverseFaceMap[oldFacei];

                if (masterFacei < 0)
                {
                    FatalErrorInFunction
                        << "Problem: should not have removed faces"
                        << " when refining."
                        << nl << "face:" << facei << abort(FatalError);
                }
                else if (masterFacei != facei)
                {
                    masterFaces.insert(masterFacei);
                }
            }
        }
        if (debug)
        {
            Pout<< "Found " << masterFaces.size() << " split faces " << endl;
        }

        HashTable<surfaceScalarField*> fluxes
        (
            lookupClass<surfaceScalarField>()
        );
        forAllIter(HashTable<surfaceScalarField*>, fluxes, iter)
        {
            if (!correctFluxes_.found(iter.key()))
            {
                WarningInFunction
                    << "Cannot find surfaceScalarField " << iter.key()
                    << " in user-provided flux mapping table "
                    << correctFluxes_ << endl
                    << "    The flux mapping table is used to recreate the"
                    << " flux on newly created faces." << endl
                    << "    Either add the entry if it is a flux or use ("
                    << iter.key() << " none) to suppress this warning."
                    << endl;
                continue;
            }

            const word& UName = correctFluxes_[iter.key()];

            if (UName == "none")
            {
                continue;
            }

            if (UName == "NaN")
            {
                Pout<< "Setting surfaceScalarField " << iter.key()
                    << " to NaN" << endl;

```

```
                surfaceScalarField& phi = *iter();

                sigFpe::fillNan(phi.primitiveFieldRef());

                continue;
            }

            if (debug)
            {
                Pout<< "Mapping flux " << iter.key()
                    << " using interpolated flux " << UName
                    << endl;
            }

            surfaceScalarField& phi = *iter();
            const surfaceScalarField phiU
            (
                fvc::interpolate
                (
                    lookupObject<volVectorField>(UName)
                )
              & Sf()
            );

            // Recalculate new internal faces.
            for (label facei = 0; facei < nInternalFaces(); facei++)
            {
                label oldFacei = faceMap[facei];

                if (oldFacei == -1)
                {
                    // Inflated/appended
                    phi[facei] = phiU[facei];
                }
                else if (reverseFaceMap[oldFacei] != facei)
                {
                    // face-from-masterface
                    phi[facei] = phiU[facei];
                }
            }

            // Recalculate new boundary faces.
            surfaceScalarField::Boundary& phiBf =
                phi.boundaryFieldRef();
            forAll(phiBf, patchi)
            {
                fvsPatchScalarField& patchPhi = phiBf[patchi];
                const fvsPatchScalarField& patchPhiU =
                    phiU.boundaryField()[patchi];

                label facei = patchPhi.patch().start();

                forAll(patchPhi, i)
                {
                    label oldFacei = faceMap[facei];

                    if (oldFacei == -1)
                    {
                        // Inflated/appended
                        patchPhi[i] = patchPhiU[i];
                    }
                    else if (reverseFaceMap[oldFacei] != facei)
                    {
                        // face-from-masterface
                        patchPhi[i] = patchPhiU[i];
                    }

                    facei++;
```

```
                }
            }

            // Update master faces
            forAllConstIter(labelHashSet, masterFaces, iter)
            {
                label facei = iter.key();

                if (isInternalFace(facei))
                {
                    phi[facei] = phiU[facei];
                }
                else
                {
                    label patchi = boundaryMesh().whichPatch(facei);
                    label i = facei - boundaryMesh()[patchi].start();

                    const fvsPatchScalarField& patchPhiU =
                        phiU.boundaryField()[patchi];

                    fvsPatchScalarField& patchPhi = phiBf[patchi];

                    patchPhi[i] = patchPhiU[i];
                }
            }
        }
    }


    // Update numbering of cells/vertices.
    meshCutter_.updateMesh(map);

    // Update numbering of protectedCells_
    if (protectedCells_.size())
    {
        PackedBoolList newProtectedCell(nCells());

        forAll(newProtectedCell, celli)
        {
            label oldCelli = map().cellMap()[celli];
            newProtectedCell.set(celli, protectedCells_.get(oldCelli));
        }
        protectedCells_.transfer(newProtectedCell);
    }

    // Debug: Check refinement levels (across faces only)
    meshCutter_.checkRefinementLevels(-1, labelList(0));

    return map;
}


Foam::autoPtr<Foam::mapPolyMesh>
Foam::dynamicRefineFvMesh::unrefine
(
    const labelList& splitPoints
)
{
    polyTopoChange meshMod(*this);

    // Play refinement commands into mesh changer.
    meshCutter_.setUnrefinement(splitPoints, meshMod);


    // Save information on faces that will be combined
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Find the faceMidPoints on cells to be combined.
```

```
    // for each face resulting of split of face into four store the
    // midpoint
    Map<label> faceToSplitPoint(3*splitPoints.size());

    {
        forAll(splitPoints, i)
        {
            label pointi = splitPoints[i];

            const labelList& pEdges = pointEdges()[pointi];

            forAll(pEdges, j)
            {
                label otherPointi = edges()[pEdges[j]].otherVertex(pointi);

                const labelList& pFaces = pointFaces()[otherPointi];

                forAll(pFaces, pFacei)
                {
                    faceToSplitPoint.insert(pFaces[pFacei], otherPointi);
                }
            }
        }
    }


    // Change mesh and generate map.
    // autoPtr<mapPolyMesh> map = meshMod.changeMesh(*this, true);
    autoPtr<mapPolyMesh> map = meshMod.changeMesh(*this, false);

    Info<< "Unrefined from "
        << returnReduce(map().nOldCells(), sumOp<label>())
        << " to " << globalData().nTotalCells() << " cells."
        << endl;

    // Update fields
    updateMesh(map);

    // Correct the flux for modified faces.
    {
        const labelList& reversePointMap = map().reversePointMap();
        const labelList& reverseFaceMap = map().reverseFaceMap();

        HashTable<surfaceScalarField*> fluxes
        (
            lookupClass<surfaceScalarField>()
        );
        forAllIter(HashTable<surfaceScalarField*>, fluxes, iter)
        {
            if (!correctFluxes_.found(iter.key()))
            {
                WarningInFunction
                    << "Cannot find surfaceScalarField " << iter.key()
                    << " in user-provided flux mapping table "
                    << correctFluxes_ << endl
                    << "    The flux mapping table is used to recreate the"
                    << " flux on newly created faces." << endl
                    << "    Either add the entry if it is a flux or use ("
                    << iter.key() << " none) to suppress this warning."
                    << endl;
                continue;
            }

            const word& UName = correctFluxes_[iter.key()];

            if (UName == "none")
            {
                continue;
```

```
            }

            if (debug)
            {
                Info<< "Mapping flux " << iter.key()
                    << " using interpolated flux " << UName
                    << endl;
            }

            surfaceScalarField& phi = *iter();
            surfaceScalarField::Boundary& phiBf =
                phi.boundaryFieldRef();

            const surfaceScalarField phiU
            (
                fvc::interpolate
                (
                    lookupObject<volVectorField>(UName)
                )
              & Sf()
            );


            forAllConstIter(Map<label>, faceToSplitPoint, iter)
            {
                label oldFacei = iter.key();
                label oldPointi = iter();

                if (reversePointMap[oldPointi] < 0)
                {
                    // midpoint was removed. See if face still exists.
                    label facei = reverseFaceMap[oldFacei];

                    if (facei >= 0)
                    {
                        if (isInternalFace(facei))
                        {
                            phi[facei] = phiU[facei];
                        }
                        else
                        {
                            label patchi = boundaryMesh().whichPatch(facei);
                            label i = facei - boundaryMesh()[patchi].start();

                            const fvsPatchScalarField& patchPhiU =
                                phiU.boundaryField()[patchi];
                            fvsPatchScalarField& patchPhi = phiBf[patchi];
                            patchPhi[i] = patchPhiU[i];
                        }
                    }
                }
            }
        }
    }


    // Update numbering of cells/vertices.
    meshCutter_.updateMesh(map);

    // Update numbering of protectedCells_
    if (protectedCells_.size())
    {
        PackedBoolList newProtectedCell(nCells());

        forAll(newProtectedCell, celli)
        {
            label oldCelli = map().cellMap()[celli];
            if (oldCelli >= 0)
```

```
            {
                newProtectedCell.set(celli, protectedCells_.get(oldCelli));
            }
        }
        protectedCells_.transfer(newProtectedCell);
    }

    // Debug: Check refinement levels (across faces only)
    meshCutter_.checkRefinementLevels(-1, labelList(0));

    return map;
}


const Foam::cellZone& Foam::dynamicRefineFvMesh::findCellZone
(
    const word& cellZoneName
) const
{
    const label cellZoneID = cellZones().findZoneID(cellZoneName);
    bool cellZoneFound = (cellZoneID != -1);
    reduce(cellZoneFound, orOp<bool>());
    if (!cellZoneFound)
    {
        FatalErrorInFunction
            << "cannot find cellZone " << cellZoneName
            << exit(FatalError);
    }

    return cellZones()[cellZoneID];
}


Foam::scalarField
Foam::dynamicRefineFvMesh::cellToPoint(const scalarField& vFld) const
{
    scalarField pFld(nPoints());

    forAll(pointCells(), pointi)
    {
        const labelList& pCells = pointCells()[pointi];

        scalar sum = 0.0;
        forAll(pCells, i)
        {
            sum += vFld[pCells[i]];
        }
        pFld[pointi] = sum/pCells.size();
    }
    return pFld;
}


Foam::scalarField Foam::dynamicRefineFvMesh::error
(
    const scalarField& fld,
    const scalar minLevel,
    const scalar maxLevel
) const
{
    scalarField c(fld.size(), -1);

    forAll(c, celli)
    {
        scalar err = min(fld[celli] - minLevel, maxLevel - fld[celli]);

        if (err >= 0)
        {
```

```
            c[celli] = err;
        }
    }

    return c;
}


Foam::scalarField Foam::dynamicRefineFvMesh::error
(
    const scalarField& fld,
    const labelList& cells,
    const scalar minLevel,
    const scalar maxLevel
) const
{
    scalarField c(fld.size(), -1);

    forAll(cells, i)
    {
        const label celli = cells[i];

        scalar err = min(fld[celli] - minLevel, maxLevel - fld[celli]);

        if (err >= 0)
        {
            c[celli] = err;
        }
    }

    return c;
}


void Foam::dynamicRefineFvMesh::selectRefineCandidates
(
    PackedBoolList& candidateCells,
    const scalar lowerRefineLevel,
    const scalar upperRefineLevel,
    const scalar maxRefinement,
    const scalarField& vFld
) const
{
    // Get error per cell. Is -1 (not to be refined) to >0 (to be refined,
    // higher more desirable to be refined).
    const scalarField cellError
    (
        error(vFld, lowerRefineLevel, upperRefineLevel)
    );

    const labelList& cellLevel = meshCutter_.cellLevel();

    // Mark cells that are candidates for refinement.
    forAll(cellError, celli)
    {
        if
        (
            cellLevel[celli] < maxRefinement
         && cellError[celli] > 0
        )
        {
            candidateCells.set(celli, 1);
        }
    }
}


void Foam::dynamicRefineFvMesh::selectRefineCandidates
```

```
(
    PackedBoolList& candidateCells,
    const scalar lowerRefineLevel,
    const scalar upperRefineLevel,
    const scalar maxRefinement,
    const scalarField& vFld,
    const labelList& cells
) const
{
    // Get error per cell. Is -1 (not to be refined) to >0 (to be refined,
    // higher more desirable to be refined).
    const scalarField cellError
    (
        error(vFld, cells, lowerRefineLevel, upperRefineLevel)
    );

    const labelList& cellLevel = meshCutter_.cellLevel();

    // Mark cells that are candidates for refinement.
    forAll(cellError, celli)
    {
        if
        (
            cellLevel[celli] < maxRefinement
         && cellError[celli] > 0
        )
        {
            candidateCells.set(celli, 1);
        }
    }
}


Foam::scalar Foam::dynamicRefineFvMesh::selectRefineCandidates
(
    PackedBoolList& candidateCells,
    const dictionary& refineDict
) const
{
    const word fieldName(refineDict.lookup("field"));

    const volScalarField& vFld = lookupObject<volScalarField>(fieldName);

    const scalar lowerRefineLevel =
        refineDict.lookup<scalar>("lowerRefineLevel");
    const scalar upperRefineLevel =
        refineDict.lookup<scalar>("upperRefineLevel");

    const label maxRefinement = refineDict.lookup<label>("maxRefinement");

    if (maxRefinement <= 0)
    {
        FatalErrorInFunction
            << "Illegal maximum refinement level " << maxRefinement << nl
            << "The maxCells setting in the dynamicMeshDict should"
            << " be > 0." << nl
            << exit(FatalError);
    }

    if (refineDict.found("cellZone"))
    {
        // Determine candidates for refinement (looking at field only)
        selectRefineCandidates
        (
            candidateCells,
            lowerRefineLevel,
            upperRefineLevel,
            maxRefinement,
```

```
            vFld,
            findCellZone(refineDict.lookup("cellZone"))
        );
    }
    else
    {
        // Determine candidates for refinement (looking at field only)
        selectRefineCandidates
        (
            candidateCells,
            lowerRefineLevel,
            upperRefineLevel,
            maxRefinement,
            vFld
        );
    }

    return maxRefinement;
}


Foam::labelList Foam::dynamicRefineFvMesh::selectRefineCells
(
    const label maxCells,
    const label maxRefinement,
    const PackedBoolList& candidateCells
) const
{
    // Every refined cell causes 7 extra cells
    label nTotToRefine = (maxCells - globalData().nTotalCells()) / 7;

    const labelList& cellLevel = meshCutter_.cellLevel();

    // Mark cells that cannot be refined since they would trigger refinement
    // of protected cells (since 2:1 cascade)
    PackedBoolList unrefineableCells;
    calculateProtectedCells(unrefineableCells);

    // Count current selection
    label nLocalCandidates = count(candidateCells, 1);
    label nCandidates = returnReduce(nLocalCandidates, sumOp<label>());

    // Collect all cells
    DynamicList<label> candidates(nLocalCandidates);

    if (nCandidates < nTotToRefine)
    {
        forAll(candidateCells, celli)
        {
            if
            (
                candidateCells.get(celli)
             && (
                    unrefineableCells.empty()
                 || !unrefineableCells.get(celli)
                )
            )
            {
                candidates.append(celli);
            }
        }
    }
    else
    {
        // Sort by error? For now just truncate.
        for (label level = 0; level < maxRefinement; level++)
        {
            forAll(candidateCells, celli)
```

```
            {
                if
                (
                    cellLevel[celli] == level
                 && candidateCells.get(celli)
                 && (
                        unrefineableCells.empty()
                     || !unrefineableCells.get(celli)
                    )
                )
                {
                    candidates.append(celli);
                }
            }

            if (returnReduce(candidates.size(), sumOp<label>()) > nTotToRefine)
            {
                break;
            }
        }
    }

    // Guarantee 2:1 refinement after refinement
    labelList consistentSet
    (
        meshCutter_.consistentRefinement
        (
            candidates.shrink(),
            true               // Add to set to guarantee 2:1
        )
    );

    Info<< "Selected " << returnReduce(consistentSet.size(), sumOp<label>())
        << " cells for refinement out of " << globalData().nTotalCells()
        << "." << endl;

    return consistentSet;
}


void Foam::dynamicRefineFvMesh::selectUnrefineCandidates
(
    boolList& unrefineCandidates,
    const volScalarField& vFld,
    const scalar unrefineLevel
) const
{
    forAll(pointCells(), pointi)
    {
        const labelList& pCells = pointCells()[pointi];

        scalar maxVal = -great;
        forAll(pCells, i)
        {
            maxVal = max(maxVal, vFld[pCells[i]]);
        }

        unrefineCandidates[pointi] =
            unrefineCandidates[pointi] && maxVal < unrefineLevel;
    }
}


void Foam::dynamicRefineFvMesh::selectUnrefineCandidates
(
    boolList& unrefineCandidates,
    const volScalarField& vFld,
    const cellZone& cZone,
```

```
    const scalar unrefineLevel
) const
{
    const Map<label>& zoneMap(cZone.lookupMap());

    forAll(pointCells(), pointi)
    {
        const labelList& pCells = pointCells()[pointi];

        scalar maxVal = -great;
        forAll(pCells, i)
        {
            if (zoneMap.found(pCells[i]))
            {
                maxVal = max(maxVal, vFld[pCells[i]]);
            }
        }

        unrefineCandidates[pointi] =
            unrefineCandidates[pointi] && maxVal < unrefineLevel;
    }
}


void Foam::dynamicRefineFvMesh::selectUnrefineCandidates
(
    boolList& unrefineCandidates,
    const dictionary& refineDict
) const
{
    if (refineDict.found("unrefineLevel"))
    {
        const word fieldName(refineDict.lookup("field"));
        const volScalarField& vFld
        (
            lookupObject<volScalarField>(fieldName)
        );

        const scalar unrefineLevel =
            refineDict.lookup<scalar>("unrefineLevel");

        if (refineDict.found("cellZone"))
        {
            selectUnrefineCandidates
            (
                unrefineCandidates,
                vFld,
                findCellZone(refineDict.lookup("cellZone")),
                unrefineLevel
            );
        }
        else
        {
            selectUnrefineCandidates
            (
                unrefineCandidates,
                vFld,
                unrefineLevel
            );
        }
    }
}


Foam::labelList Foam::dynamicRefineFvMesh::selectUnrefinePoints
(
    const PackedBoolList& markedCell,
    const boolList& unrefineCandidates
```

```
) const
{
    // All points that can be unrefined
    const labelList splitPoints(meshCutter_.getSplitPoints());

    DynamicList<label> newSplitPoints(splitPoints.size());

    forAll(splitPoints, i)
    {
        label pointi = splitPoints[i];

        if (unrefineCandidates[pointi])
        {
            // Check that all cells are not marked
            const labelList& pCells = pointCells()[pointi];

            bool hasMarked = false;

            forAll(pCells, pCelli)
            {
                if (markedCell.get(pCells[pCelli]))
                {
                    hasMarked = true;
                    break;
                }
            }

            if (!hasMarked)
            {
                newSplitPoints.append(pointi);
            }
        }
    }


    newSplitPoints.shrink();

    // Guarantee 2:1 refinement after unrefinement
    labelList consistentSet
    (
        meshCutter_.consistentUnrefinement
        (
            newSplitPoints,
            false
        )
    );
    Info<< "Selected " << returnReduce(consistentSet.size(), sumOp<label>())
        << " split points out of a possible "
        << returnReduce(splitPoints.size(), sumOp<label>())
        << "." << endl;

    return consistentSet;
}


void Foam::dynamicRefineFvMesh::extendMarkedCells
(
    PackedBoolList& markedCell
) const
{
    // Mark faces using any marked cell
    boolList markedFace(nFaces(), false);

    forAll(markedCell, celli)
    {
        if (markedCell.get(celli))
        {
            const cell& cFaces = cells()[celli];
```

```

            forAll(cFaces, i)
            {
                markedFace[cFaces[i]] = true;
            }
        }
    }

    syncTools::syncFaceList(*this, markedFace, orEqOp<bool>());

    // Update cells using any markedFace
    for (label facei = 0; facei < nInternalFaces(); facei++)
    {
        if (markedFace[facei])
        {
            markedCell.set(faceOwner()[facei], 1);
            markedCell.set(faceNeighbour()[facei], 1);
        }
    }
    for (label facei = nInternalFaces(); facei < nFaces(); facei++)
    {
        if (markedFace[facei])
        {
            markedCell.set(faceOwner()[facei], 1);
        }
    }
}


void Foam::dynamicRefineFvMesh::checkEightAnchorPoints
(
    PackedBoolList& protectedCell,
    label& nProtected
) const
{
    const labelList& cellLevel = meshCutter_.cellLevel();
    const labelList& pointLevel = meshCutter_.pointLevel();

    labelList nAnchorPoints(nCells(), 0);

    forAll(pointLevel, pointi)
    {
        const labelList& pCells = pointCells(pointi);

        forAll(pCells, pCelli)
        {
            label celli = pCells[pCelli];

            if (pointLevel[pointi] <= cellLevel[celli])
            {
                // Check if cell has already 8 anchor points -> protect cell
                if (nAnchorPoints[celli] == 8)
                {
                    if (protectedCell.set(celli, true))
                    {
                        nProtected++;
                    }
                }

                if (!protectedCell[celli])
                {
                    nAnchorPoints[celli]++;
                }
            }
        }
    }


```

```
    forAll(protectedCell, celli)
    {
        if (!protectedCell[celli] && nAnchorPoints[celli] != 8)
        {
            protectedCell.set(celli, true);
            nProtected++;
        }
    }
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::dynamicRefineFvMesh::dynamicRefineFvMesh(const IOobject& io)
:
    dynamicFvMesh(io),
    meshCutter_(*this),
    dumpLevel_(false),
    nRefinementIterations_(0),
    protectedCells_(nCells(), 0)
{
    // Read static part of dictionary
    readDict();

    const labelList& cellLevel = meshCutter_.cellLevel();
    const labelList& pointLevel = meshCutter_.pointLevel();

    // Set cells that should not be refined.
    // This is currently any cell which does not have 8 anchor points or
    // uses any face which does not have 4 anchor points.
    // Note: do not use cellPoint addressing

    // Count number of points <= cellLevel
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    labelList nAnchors(nCells(), 0);

    label nProtected = 0;

    forAll(pointCells(), pointi)
    {
        const labelList& pCells = pointCells()[pointi];

        forAll(pCells, i)
        {
            label celli = pCells[i];

            if (!protectedCells_.get(celli))
            {
                if (pointLevel[pointi] <= cellLevel[celli])
                {
                    nAnchors[celli]++;

                    if (nAnchors[celli] > 8)
                    {
                        protectedCells_.set(celli, 1);
                        nProtected++;
                    }
                }
            }
        }
    }


    // Count number of points <= faceLevel
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Bit tricky since proc face might be one more refined than the owner since
    // the coupled one is refined.
```

```

    {
        labelList neiLevel(nFaces());

        for (label facei = 0; facei < nInternalFaces(); facei++)
        {
            neiLevel[facei] = cellLevel[faceNeighbour()[facei]];
        }
        for (label facei = nInternalFaces(); facei < nFaces(); facei++)
        {
            neiLevel[facei] = cellLevel[faceOwner()[facei]];
        }
        syncTools::swapFaceList(*this, neiLevel);


        boolList protectedFace(nFaces(), false);

        forAll(faceOwner(), facei)
        {
            label faceLevel = max
            (
                cellLevel[faceOwner()[facei]],
                neiLevel[facei]
            );

            const face& f = faces()[facei];

            label nAnchors = 0;

            forAll(f, fp)
            {
                if (pointLevel[f[fp]] <= faceLevel)
                {
                    nAnchors++;

                    if (nAnchors > 4)
                    {
                        protectedFace[facei] = true;
                        break;
                    }
                }
            }
        }

        syncTools::syncFaceList(*this, protectedFace, orEqOp<bool>());

        for (label facei = 0; facei < nInternalFaces(); facei++)
        {
            if (protectedFace[facei])
            {
                protectedCells_.set(faceOwner()[facei], 1);
                nProtected++;
                protectedCells_.set(faceNeighbour()[facei], 1);
                nProtected++;
            }
        }
        for (label facei = nInternalFaces(); facei < nFaces(); facei++)
        {
            if (protectedFace[facei])
            {
                protectedCells_.set(faceOwner()[facei], 1);
                nProtected++;
            }
        }

        // Also protect any cells that are less than hex
        forAll(cells(), celli)
        {
```

```
            const cell& cFaces = cells()[celli];

            if (cFaces.size() < 6)
            {
                if (protectedCells_.set(celli, 1))
                {
                    nProtected++;
                }
            }
            else
            {
                forAll(cFaces, cFacei)
                {
                    if (faces()[cFaces[cFacei]].size() < 4)
                    {
                        if (protectedCells_.set(celli, 1))
                        {
                            nProtected++;
                        }
                        break;
                    }
                }
            }
        }

        // Check cells for 8 corner points
        checkEightAnchorPoints(protectedCells_, nProtected);
    }

    if (returnReduce(nProtected, sumOp<label>()) == 0)
    {
        protectedCells_.clear();
    }
    else
    {

        cellSet protectedCells(*this, "protectedCells", nProtected);
        forAll(protectedCells_, celli)
        {
            if (protectedCells_[celli])
            {
                protectedCells.insert(celli);
            }
        }

        Info<< "Detected " << returnReduce(nProtected, sumOp<label>())
            << " cells that are protected from refinement."
            << " Writing these to cellSet "
            << protectedCells.name()
            << "." << endl;

        protectedCells.write();
    }
}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

Foam::dynamicRefineFvMesh::~dynamicRefineFvMesh()
{}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

bool Foam::dynamicRefineFvMesh::update()
{
    // Re-read dictionary. Chosen since usually -small so trivial amount
    // of time compared to actual refinement. Also very useful to be able
```

```
    // to modify on-the-fly.
    const dictionary refineDict
    (
        dynamicMeshDict().optionalSubDict(typeName + "Coeffs")
    );

    label refineInterval = refineDict.lookup<label>("refineInterval");

    bool hasChanged = false;

    if (refineInterval == 0)
    {
        topoChanging(hasChanged);

        return false;
    }
    else if (refineInterval < 0)
    {
        FatalErrorInFunction
            << "Illegal refineInterval " << refineInterval << nl
            << "The refineInterval setting in the dynamicMeshDict should"
            << " be >= 1." << nl
            << exit(FatalError);
    }

    // Note: cannot refine at time 0 since no V0 present since mesh not
    //       moved yet.

    if (time().timeIndex() > 0 && time().timeIndex() % refineInterval == 0)
    {
        label maxCells = refineDict.lookup<label>("maxCells");

        if (maxCells <= 0)
        {
            FatalErrorInFunction
                << "Illegal maximum number of cells " << maxCells << nl
                << "The maxCells setting in the dynamicMeshDict should"
                << " be > 0." << nl
                << exit(FatalError);
        }

        const label nBufferLayers =
            refineDict.lookup<label>("nBufferLayers");

        // Cells marked for refinement or otherwise protected from unrefinement.
        PackedBoolList refineCells(nCells());

        label maxRefinement = 0;

        if (refineDict.isDict("refinementRegions"))
        {
            const dictionary& refinementRegions
            (
                refineDict.subDict("refinementRegions")
            );

            forAllConstIter(dictionary, refinementRegions, iter)
            {
                maxRefinement = max
                (
                    selectRefineCandidates
                    (
                        refineCells,
                        refinementRegions.subDict(iter().keyword())
                    ),
                    maxRefinement
                );
            }
```

```
        }
        else
        {
            maxRefinement = selectRefineCandidates(refineCells, refineDict);
        }

        if (globalData().nTotalCells() < maxCells)
        {
            // Select subset of candidates. Take into account max allowable
            // cells, refinement level, protected cells.
            labelList cellsToRefine
            (
                selectRefineCells
                (
                    maxCells,
                    maxRefinement,
                    refineCells
                )
            );

            label nCellsToRefine = returnReduce
            (
                cellsToRefine.size(), sumOp<label>()
            );

            if (nCellsToRefine > 0)
            {
                // Refine/update mesh and map fields
                autoPtr<mapPolyMesh> map = refine(cellsToRefine);

                // Update refineCells. Note that some of the marked ones have
                // not been refined due to constraints.
                {
                    const labelList& cellMap = map().cellMap();
                    const labelList& reverseCellMap = map().reverseCellMap();

                    PackedBoolList newRefineCell(cellMap.size());

                    forAll(cellMap, celli)
                    {
                        label oldCelli = cellMap[celli];

                        if (oldCelli < 0)
                        {
                            newRefineCell.set(celli, 1);
                        }
                        else if (reverseCellMap[oldCelli] != celli)
                        {
                            newRefineCell.set(celli, 1);
                        }
                        else
                        {
                            newRefineCell.set(celli, refineCells.get(oldCelli));
                        }
                    }
                    refineCells.transfer(newRefineCell);
                }

                // Extend with a buffer layer to prevent neighbouring points
                // being unrefined.
                for (label i = 0; i < nBufferLayers; i++)
                {
                    extendMarkedCells(refineCells);
                }

                hasChanged = true;
            }
        }
```

```
        boolList unrefineCandidates(nPoints(), true);

        if (refineDict.isDict("refinementRegions"))
        {
            const dictionary& refinementRegions
            (
                refineDict.subDict("refinementRegions")
            );

            forAllConstIter(dictionary, refinementRegions, iter)
            {
                selectUnrefineCandidates
                (
                    unrefineCandidates,
                    refinementRegions.subDict(iter().keyword())
                );
            }
        }
        else
        {
            selectUnrefineCandidates
            (
                unrefineCandidates,
                refineDict
            );
        }

        {
            // Select unrefineable points that are not marked in refineCells
            labelList pointsToUnrefine
            (
                selectUnrefinePoints
                (
                    refineCells,
                    unrefineCandidates
                )
            );

            label nSplitPoints = returnReduce
            (
                pointsToUnrefine.size(),
                sumOp<label>()
            );

            if (nSplitPoints > 0)
            {
                // Refine/update mesh
                unrefine(pointsToUnrefine);

                hasChanged = true;
            }
        }


        if ((nRefinementIterations_ % 10) == 0)
        {
            // Compact refinement history occasionally (how often?).
            // Unrefinement causes holes in the refinementHistory.
            const_cast<refinementHistory&>(meshCutter().history()).compact();
        }
        nRefinementIterations_++;
    }

    topoChanging(hasChanged);
    if (hasChanged)
    {
        // Reset moving flag (if any). If not using inflation we'll not move,
```

```
        // if are using inflation any follow on movePoints will set it.
        moving(false);
    }

    return hasChanged;
}


bool Foam::dynamicRefineFvMesh::writeObject
(
    IOstream::streamFormat fmt,
    IOstream::versionNumber ver,
    IOstream::compressionType cmp,
    const bool write
) const
{
    // Force refinement data to go to the current time directory.
    const_cast<hexRef8&>(meshCutter_).setInstance(time().timeName());

    bool writeOk =
    (
        dynamicFvMesh::writeObject(fmt, ver, cmp, write)
     && meshCutter_.write(write)
    );

    if (dumpLevel_)
    {
        volScalarField scalarCellLevel
        (
            IOobject
            (
                "cellLevel",
                time().timeName(),
                *this,
                IOobject::NO_READ,
                IOobject::AUTO_WRITE,
                false
            ),
            *this,
            dimensionedScalar(dimless, 0)
        );

        const labelList& cellLevel = meshCutter_.cellLevel();

        forAll(cellLevel, celli)
        {
            scalarCellLevel[celli] = cellLevel[celli];
        }

        writeOk = writeOk && scalarCellLevel.write();
    }

    return writeOk;
}


// ************************************************************************* //
```